Perl语言入门

Randal L.Schwartz, brian d foy &
Tom Phoenix 著
盛春译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权东南大学出版社出版

东南大学出版社

图书在版编目（CIP）数据

Perl语言入门：第6版/（美）施瓦茨（Schwartz, R.L.),（美）福瓦（Foy, B.D.),（美）菲尼克斯（Phoenix, T.）著；盛春译. —南京：东南大学出版社，2012.3

书名原文：Learning Perl，6E

ISBN 978-7-5641-3372-6

I. ① P… II. ①施… ②福… ③菲… ④盛… III. ① Perl语言－程序设计 IV. ① TP312

中国版本图书馆CIP数据核字（2012）第036637号

江苏省版权局著作权合同登记

图字：10-2011-417号

©2011 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Southeast University Press, 2012. Authorized translation of the English edition, 2011 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由O'Reilly Media, Inc. 出版2011。

简体中文版由东南大学出版社出版2012。英文原版的翻译得到O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者 —— O'Reilly Media, Inc. 的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

Perl语言入门 第6版

出版发行：东南大学出版社
地　　址：南京四牌楼2号　　邮编：210096
出 版 人：江建中
网　　址：http://www.seupress.com
电子邮件：press@seupress.com
印　　刷：扬中市印刷有限公司
开　　本：787毫米×980毫米　16开本
印　　张：24印张
字　　数：470千字
版　　次：2012年3月第1版
印　　次：2012年3月第1次印刷
书　　号：ISBN 978-7-5641-3372-6
印　　数：1~4000册
定　　价：62.00元（册）

本社图书若有印装质量问题，请直接与读者服务部联系。电话（传真）：025-83792328

O'Reilly Media, Inc. 介绍

O'Reilly Media, Inc. 是世界上在 UNIX、X、Internet 和其他开放系统图书领域具有领导地位的出版公司，同时是联机出版的先锋。

从最畅销的《The Whole Internet User's Guide & Catalog》（被纽约公共图书馆评为二十世纪最重要的50本书之一）到GNN（最早的Internet门户和商业网站），再到WebSite（第一个桌面PC的Web服务器软件），O'Reilly Media, Inc.一直处于Internet发展的最前沿。

许多书店的反馈表明，O'Reilly Media, Inc. 是最稳定的计算机图书出版商——每一本书都一版再版。与大多数计算机图书出版商相比，O'Reilly Media, Inc. 具有深厚的计算机专业背景，这使得 O'Reilly Media, Inc. 形成了一个非常不同于其他出版商的出版方针。O'Reilly Media, Inc. 所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O'Reilly Media, Inc. 还有许多固定的作者群体——他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O'Reilly Media, Inc. 依靠他们及时地推出图书。因为 O'Reilly Media, Inc. 紧密地与计算机业界联系着，所以 O'Reilly Media, Inc. 知道市场上真正需要什么图书。

大家网
TopSage.com

目录

第一章 简介..10

问题与答案..10

这本书适合你吗？..10

为何有这么多的脚注？..11

关于习题和解答？..12

习题前标的数字是什么意思？..13

如果我是Perl讲师？..13

“Perl”这个词表示什么意思？..13

Larry为什么要创造Perl？..13

Larry干吗不用其他语言？..14

Perl算容易，还是算难？..15

Perl怎么会这么流行？..16

现在的Perl发展得怎么样了？..16

哪些事情最适合用Perl来做？..17

哪些事情不适合用Perl来做？..18

如何取得Perl？..18

CPAN是什么？..19

如何得到Perl的技术支持？..19

还有别的技术支持方式吗？..20

如果发现Perl有bug，我该怎么办？..21

我该怎么编写Perl程序？..22

一个简单的程序..22

程序里写的是什么？......24
我该如何编译Perl程序？......26
走马观花......27
习题......28

第二章 标量数据......30
数字......30
所有数字的内部格式都相同......30
浮点数直接量......31
整数直接量......31
非十进制整数的直接量......32
数字操作符......33
字符串......33
单引号内的字符串直接量......34
双引号内的字符串直接量......34
字符串操作符......36
数字与字符串之间的自动转换......36
Perl的内置警告信息......37
标量变量......39
给变量取个好名字......40
标量的赋值......40
双目赋值操作符......41
用print输出结果......41
字符串中的标量变量内插......42
借助代码点创建字符......43
操作符的优先级与结合性......44
比较操作符......46
if控制结构......47
布尔值......47
获取用户输入......48
chomp操作符......49
while控制结构......50

undef值 50
defined函数 51
习题 52

第三章 列表与数组 53
访问数组中的元素 54
特殊的数组索引 55
列表直接量 56
qw简写 56
列表的赋值 58
pop和push操作符 59
shift和unshift操作符 61
splice操作符 61
字符串中的数组内插 62
foreach控制结构 63
Perl最喜欢用的默认变量：$_ 64
reverse操作符 65
sort操作符 65
each操作符 66
标量上下文与列表上下文 66
在标量上下文中使用产生列表的表达式 68
在列表上下文中使用产生标量的表达式 69
强制指定标量上下文 70
列表上下文中的<STDIN> 70
习题 72

第四章 子程序 73
定义子程序 73
调用子程序 74
返回值 75
参数 76

子程序中的私有变量......78
变长参数列表......79
改进的&max子程序......79
空参数列表......80
关于词法（my）变量......81
use strict编译指令......82
return操作符......84
省略与号......85
非标量返回值......86
持久化私有变量......87
习题......88

第五章 输入与输出......90
读取标准输入......90
来自钻石操作符的输入......92
调用参数......94
输出到标准输出......95
用printf格式化输出......98
数组和printf......100
文件句柄......100
打开文件句柄......102
以二进制方式读写文件句柄......105
有问题的文件句柄......106
关闭文件句柄......106
用die处理致命错误......107
用warn输出警告信息......109
自动检测致命错误......109
使用文件句柄......110
改变默认的文件输出句柄......111
重新打开标准文件句柄......111
用say来输出......112

标量变量中的文件句柄 113
习题 115

第六章 哈希 116

什么是哈希？ 116
为何使用哈希？ 118
访问哈希元素 119
访问整个哈希 121
哈希赋值 122
胖箭头 123
哈希函数 124
keys和values函数 124
each函数 125
哈希的典型应用 126
exists函数 127
delete函数 127
哈希元素内插 128
%ENV哈希 128
习题 129

第七章 漫游正则表达式王国 130

什么是正则表达式？ 130
使用简单模式 131
Unicode属性 132
关于元字符 133
简单的量词 133
模式分组 134
择一匹配 137
字符集 137
字符集的简写 138
反义简写 141

习题 141

第八章 用正则表达式进行匹配 143

用m//进行匹配 143
模式匹配修饰符 144
用/i进行大小写无关的匹配 144
用/s匹配任意字符 144
用/x加入空白符 145
组合选项修饰符 145
选择一种字符解释方式 146
其他选项 148
锚位 148
单词锚位 150
绑定操作符=~ 151
模式中的内插 152
捕获变量 153
捕获变量的存续期 154
不捕获模式 155
命名捕获 156
自动捕获变量 158
通用量词 160
优先级 161
优先级范例 162
还有更多 162
模式测试程序 162
习题 163

第九章 用正则表达式处理文本 165

用s///进行替换 165
用/g进行全局替换 166
不同的定界符 167

可用替换修饰符 167
绑定操作符 167
无损替换 167
大小写转换 168
split操作符 169
join函数 170
列表上下文中的m// 171
更强大的正则表达式 172
非贪婪量词 172
跨行的模式匹配 174
一次更新多个文件 175
从命令行直接编辑 177
习题 178

第十章 其他控制结构 180
unless控制结构 180
伴随unless的else子句 181
until控制结构 181
表达式修饰符 182
裸块控制结构 183
elsif子句 184
自增与自减 185
自增的值 186
for控制结构 187
foreach和for间的秘密关系 189
循环控制 190
last操作符 190
next操作符 191
redo操作符 192
带标签的块 193
条件操作符?: 194

逻辑操作符 195
短路操作符的值 196
定义或操作符 196
使用部分求值操作符的控制结构 197
习题 199

第十一章 Perl模块 201

寻找模块 201
安装模块 202
安装到自己的目录 204
使用简易模块 205
File::Basename模块 206
仅选用模块中的部分函数 207
File::Spec模块 208
Path::Class模块 210
CGI.pm模块 210
数据库和DBI模块 211
处理日期和时间的模块 212
习题 214

第十二章 文件测试 215

文件测试操作符 215
测试同一文件的多项属性 220
栈式文件测试操作符 221
stat和lstat函数 222
localtime函数 224
按位运算操作符 225
使用位字符串 226
习题 226

第十三章 目录操作 228

在目录树中移动 228
文件名通配 229
文件名通配的另一种语法 230
目录句柄 231
递归访问目录 233
文件和目录的操作 234
删除文件 234
重命名文件 236
链接与文件 237
创建和删除目录 242
修改权限 244
修改隶属关系 244
修改时间戳 245
习题 245

第十四章 字符串与排序 247

用index查找子字符串 247
用substr操作子字符串 248
用sprintf格式化字符串 250
  用sprintf格式化金额数字 250
  非十进制数字字符串的转换 252
高级排序 252
  按哈希值排序 256
  按多个键排序 257
习题 258

第十五章 智能匹配与given-when结构 259

智能匹配操作符 259
智能匹配操作的优先级 262
given语句 264

笨拙匹配 267
多个条目的when匹配 268
习题 269

第十六章 进程管理 271

system函数 271
避免使用Shell 273
环境变量 275
exec函数 275
用反引号捕获输出结果 276
在列表上下文中使用反引号 279
用IPC::System::Simple执行外部进程 280
通过文件句柄执行外部进程 281
用fork进行深入和复杂的工作 283
发送及接收信号 284
习题 287

第十七章 高级Perl技巧 288

切片 288
数组切片 290
哈希切片 292
捕获错误 293
用eval 294
更为高级的错误处理 297
autodie 299
用grep筛选列表 300
用map把列表元素变形 302
更花哨的列表工具 303
习题 305

附录 A 习题解答 307

附录 B 超越“小骆驼” 343

附录C Unicode入门 354

大家网
TopSage.com

前言

欢迎阅读《Perl语言入门》第六版，此版本顺应Perl 5.14及其后续版本的新特性而更新。不过，如果你还在用Perl 5.8的话，本书仍然是你的最佳选择。（话说回来，这个版本已经发布了好些年头了，难道你没想过升级？）

假如你正在寻找用30到45小时就能掌握Perl语言编程的最佳方式，那么你已经找到了！在后面的章节里，我们会精心安排入门指引，介绍这个在互联网中担负重任的程序语言。它也是备受全世界系统管理员、网络黑客以及聪明随性的程序员所青睐的编程语言。

我们不可能只花几小时就把Perl的全部知识传授给你，会这么保证的书大概都撒了一点谎。相对地，我们谨慎甄选了Perl中完整且实用的部分供你学习。这些材料足以编写128行以内的小程序，而大约90%的Perl程序都不需要超过这个长度的篇幅。当你准备继续深入时，建议阅读《Intermediate Perl》这本书，该书涵盖了许多本书略去不讲的深入部分。此外我们还会纳入许多相关的知识点，方便读者延伸阅读和学习。

TopSage.com

实际上每章的内容并不多，我们把它控制在一两个小时内能够读完的篇幅。每章后面都附有若干习题，帮助你巩固刚学到的知识，在附录A中还附有习题解答，供你比对参考。因此本书可说是相当适合作为“Perl入门”的课堂教材来使用。我们对此有第一手的实践经验，几乎所有内容都是逐字逐句从我们的“Learning Perl”课程教学中萃取出来的，这门招牌课程已经经过了上千名学生的实践检验。当然，除了课堂教学以外，拿来自学也是非常不错的。

虽然Perl是活生生的“Unix工具箱”，但你并不需要成为Unix大师，甚至也不必精通

Unix就可以使用本书。除非特别注明，否则我们所提到的一切都可以应用到Windows版本的ActivePerl（ActiveState公司出品）以及许许多多其他流行的Perl版本上。

阅读本书之前，虽然无需具备任何Perl基础，但我们还是衷心希望你能够对写程序的基本概念有所了解，像变量（variable）、循环（loop）、子程序（subroutine）和数组（array）以及最重要的“用你最熟悉的文本编辑器来编辑源代码”这类事情。我们不会花时间说明这些概念。有些人平生所学的第一个程序语言就是Perl，并因学习本书而获得成功。我们很高兴能看到这样的事例，但我们也无法保证每个人都能取得一样的成功。

排版约定

本书使用以下的字体惯例：

等宽字（`Constant width`）

用于方法名称（method name）、函数名称（function name）、变量（variable）、属性（attribute）以及程序代码范例。

等宽黑体字（**`Constant width bold`**）

用于表示用户输入的内容。

等宽斜体字（*`Constant width italic`*）

用于程序代码中可被替换的项目（例如：*`filename`*，表示应该将它替换成实际的文件名）。

斜体字（*Italic*）

用于正文所提到的文件名称、URL、主机名称、第一次提及的重要词汇以及命令。

脚注

一般附加在括号之内，初次（也许是第二次、第三次）阅读本书时应该略过。有一些不完全正确的用语是为了简化说明，而脚注会说明事实。通常脚注中的资料是高级主题，不会在本书其他部分讨论到。

中括号内数字（[2]）

用于每项习题开头，表示完成该题目大致需要多少分钟。当然这是我们非常粗略的估算，大抵能反映题目的难度和复杂度。该数字仅供参考。

代码范例

本书的使命是帮助你解决实际问题。我们欢迎你复制本书中的代码，或者略加改动移植

到你的项目代码中。虽然手工键入代码也不失为一种练习方式，但我们还是欢迎你直接到*http://www.learning-perl.com*下载。

基本上，你不用事先联络我们就可以使用本书所提供的程序代码及文件，除非是大量复制。举例来说，在你的程序中，若用到几段本书中的程序代码，不需要经过我们的同意；但是做成光盘发布、销售O' Reilly书中的例子，则必须经过授权。回答别人的问题时，引用本书的文字和程序代码，也不需要经过我们的同意；但在你的产品文件中，若大量加入本书的文字与程序代码，则必须经过授权。

虽非必要，但我们会十分感谢你在引用本书的内容和范例时提到出处。完整的信息通常包括书名、作者、出版商及ISBN编号。例如："Learning Perl，6th edition，by RandalL. Schwartz，Tom Phoenix, and brian d foy. Copyright 2011 Randal L.Schwartz, brian d foy, and Tom Phoenix, 978-1-449-30358-7"。如果你的情况有别于上述情形，并心有存疑的话，请给我们来信：permissions@oreilly.com。

如何联系我们

本书的内容都经过测试，尽管我们做了最大的努力，但错误和疏忽仍然是在所难免的。如果你发现有什么错误，或者是对将来的版本有什么建议，请通过下面的地址告诉我们：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）
奥莱利技术咨询（北京）有限公司

本书的网页上列出了勘误、示例和其他相关信息，可以在以下的页面进行访问：

原文书

http://www.oreilly.com/catalog/06369200184521

中文书

http://www.oreilly.com.cn/book.php?bn=9787564133726

如果想要发表关于本书的评论或询问技术问题，请发送邮件到：

bookquestions@oreilly.com

关于图书、会议、资源中心和O'Reilly网络的其他信息，请查看我们的网站：

http://www.oreilly.com
http://www.oreilly.com.cn

本书的历史

为了满足读者的好奇心，Randal在这里告诉你关于这本书的来历：

1991年我跟Larry Wall写完第一本《Perl语言编程（Programming Perl)》之后，硅谷的Taos Mountain Software公司跟我联络，要我准备一些培训课程，内容包含12节左右的课程，并训练他们的教师继续开课。我就按约写了这个课程给他们[注1]。

在课程进行了三四次之后（1991年底），有个人走到面前跟我说："不瞒你说，我真的很喜欢《Perl语言编程》这本书，但是这门课的教材更容易吸收，你真的应该写一本像这个课程的书。"这听起来像是个好机会，所以我开始认真地考虑这个点子。

我写信给Tim O'Reilly，附上了一份企划书。其中以我提供给Taos的课程纲要为基础，再根据课堂上的观察调整并修改了一些章节。这可能是有史以来我的企划书最快被接受的记录——我在15分钟后收到了Tim的回信："我们一直在等待你的第二本书。《Perl语言编程》太热销了。"接下来的一年半时间里，我就努力完成了第一版的《Perl语言入门》。

在那段时间里，我找到在硅谷以外教授Perl的机会[注2]，所以我就以正在编写阶段的《Perl语言入门》为蓝本制作了一套课程。我为许多不同的客户教课（包括我的主要签约人Intel Oregon），并利用上课所得到的反馈进一步微调本书的草稿。

注1： 在合约中，我对习题保留所有权，我希望有一天能以不同方式来使用它们，比如说我以前曾经写过的杂志专栏。习题是Taos公司的课程里唯一还能在本书中出现的东西。

注2： 我与Taos公司的合约有条独特的条款，因此不能在硅谷教授类似的课程，我也遵守了此条款很多年。

第一版在1993年11月1日[注3]问世，销售空前成功，甚至很快就追上了《Perl语言编程》的销量。

在第一版的封底上这么写着："由卓越的Perl讲师所著"。事后证明这是正确的预言。随后的几个月里，我收到来自美国各地的电子邮件，邀请我到他们那里教Perl。接下来的7年中，我的公司成了全球领先的Perl现场培训公司，我个人的飞行里程数也飙升到了百万。之后互联网的兴起更是锦上添花，许多站长都采用Perl作为内容管理、交互式CGI及网站维护的语言。

我跟Stonehenge的首席培训师兼内容经理Tom Phoenix密切合作了两年。我请他对"Llama"课程做实验，把某些东西移来移去，再打散一些内容。当他带着我们认为是最好的修订本出现时，我就联络O'Reilly，说："是该有本新书的时候了！"于是第三版就这么诞生了。

在"小骆驼书"第三版问世的两年后，我和Tom决定把一些"高级"的课程移出来成为一本独立的、专门给需要写"100到10 000行代码"的人看的书，那就是在2003年完成的"羊驼书"。

不过，在我的同事brian d foy从海湾战争回来之后，同样是讲师的他注意到了，教材必须进一步适应普通学生的需求，因此这两本书都应该适当地改写。于是他对O'Reilly推销这个想法，希望在Perl 6完成之前进行"小骆驼书"与"羊驼书"的最后一次改版（但愿如此）。而此版本的确反映了那些变动的需求。我很少需要给brian什么建议，他一向都是顶尖的作者，在写作团队里面他给人的感觉就像尽责的英国管家。

2007年12月18日，"Perl5掌门人（Perl 5 Porters）"发布了Perl 5.10，一个标志性的版本，融入了众多新特性。之前的5.8版专注于Perl的基础架构改良和Unicode支持。而最新的版本以稳固的5.8版为基础，增加了一系列崭新的特性，特别是那些取自正在开发中的Perl 6（尚未发布）的一些理念。其中某些特性，诸如正则表达式里的命名捕获，比起传统做法来要好很多，对Perl初学者来说也更容易掌握。我们未曾想过本书会有第五版，但Perl 5.10实在是太有趣了，我们无法故步不前。

此后，Perl一直处于稳定的持续发展阶段。之前我们没有机会更新本书内容到Perl 5.12，因为它的更新速度实在是太快了，我们没有跟上。而现在，我们非常高兴地引入最新的Perl 5.14的内容，与此同时，本书也已经更新到了第六版。

注3：　这个日期我记得很清楚，因为那也是我由于围绕我与Intel公司的合约的一些跟计算机有关的活动在家被逮捕的日子，后来我被判有罪。

新版更新

本书新版内容已经按照最新的Perl 5.14作了相应的修订，其中有些代码仅限于在该版本的Perl中运行。当然，在讨论Perl 5.14的特性时，我们会在行文中加以提示说明。对于那些代码片段，我们也一律用特殊的use语句加以区别，提示你使用正确的版本，比如：

```
use 5.014;  #该脚本需要Perl 5.14或更高版本才能正常运行
```

如果在代码范例中没有看到use 5.014（或者使用其他版本的相似语句）的话，就说明这段代码可以在Perl 5.8以上版本中工作。要查看你当前所用的Perl版本号，可以在命令行使用-v选项查看：

```
$ perl-v
```

下面列出一部分我们将要谈及的Perl 5.14中引入的新特性。在谈到这些新特性时，我们还会给出传统的实现方式供你对比参考：

- 我们会在行文中涉及Unicode的地方对该特性作一些说明。如果你从未接触过Unicode，不妨先读一下附录C入门中的简要介绍。不过这块硬骨头迟早是要啃的，不如现在就去读一下。Unicode的概念穿插在本书各个角落，特别是在关于标量（第二章）、输入／输入（第五章）以及排序（第十四章）的章节。
- 在关于正则表达式的一章中，我们也扩展了许多内容，包括Perl 5.14当中处理Unicode的case-folding等。另外还有新增的几个正则表达式操作符/a、/u和/l等。并且我们还会展示通过\p{}和\P{}匹配Unicod属性的新式用法。
- Perl 5.14新增了一个不篡改原始字符串的替换操作符（第九章），由此可以写出逻辑上非常自然流畅的代码。
- 自从Perl 5.10以来，智能匹配和given-when的用法与功能都已经出现了一些变化，我们会在第十五章中介绍这些新规则。
- 我们更新并扩展了“Perl模块”（第十一章）一章，补充了一些最新的动态，包括用于安装Perl模块的零配置命令行工具*cpanm*。同时我们还增加了一些模块使用的代码范例。
- 某些之前放在附录B中的“高级但没有机会演示的”特性，现在适时地移到相应的正文中介绍。特别值得注意的是，有关双箭头=>的内容已经移到介绍哈希的章节（第六章）中，有关splice的内容则移到介绍列表和数组的章节（第三章）中去了。

致谢

来自Randal

我想要感谢Stonehenge过去与现在的讲师们（Joseph Hall、Tom Phoenix、Chip Salzenberg、brian d foy与Tad McClellan），谢谢他们愿意每周到教室授课并带回自己的笔记，注明哪部分有用（以及没用），如此我们才能精准地调整本书内容。我要特别点名Tom Phoenix，我的合著者与事业伙伴，他花了大量时间改进Stonehenge的“Llama”课程，也为本书注入了最为核心的原始内容。还有brain d foy，他在第四版中担任了主要的写作任务，从而帮我完成了收件箱中无数的待办事项。

此外，我还要感谢O'Reilly的每一个人，尤其是富有耐心和眼光的前任编辑Allison Randal（不是我的亲戚，但她的姓氏拼法很赞），以及现任编辑Simon St.Laurent。还有Tim O'Reilly本人，是他让我在一开始就有了写作“小骆驼书”与“大骆驼书”这两本书的机会。

我由衷感谢过去购买本书的上千名读者，这些钱让我免于流浪街头与夜宿囚牢；感谢我班上的学生，他们把我训练成为一名更好的讲师；还有“财富一千（Fortune 1000）”上大排长龙、在过去选购我们的课程、未来也会继续捧场的客户们。

和以前一样，我得特别感谢Lyle与Jack，你们教会了我几乎所有关于写作的知识，我永远不会忘记你们。

来自Tom

我必须附和Randal对O'Reilly的每个人致上的谢意。在第三版的时候，我们的编辑是Linda Mui，她细心地指出书中过火的玩笑和脚注，当然留下来的那些也不是她的错。她与Randal在整个写作过程中不断指导我，我非常感激。第五版的编辑是Allison Randal，现在则由Simon St.Laurent担任我们新版图书的编辑。在此，我要向两位表示由衷的感谢，感谢他们付出的无可替代的贡献。

另一些跟Randal一样要感谢的是Stonehenge的讲师们，当我临时更新课程教材以尝试新的教学技巧时，你们几乎不曾抱怨过。在教学方法上，你们提出了许多我未曾想过的主意。

多年来，我在俄勒冈科学与工业博物馆（Oregon Museum of Science and Industry，OMSI）工作，而我要感谢那里的人们，他们迫使我磨炼自己的教学技巧，让我学着在每个活动、展示与讲解中插入一两个笑话。

谢谢新闻组上的伙伴们，你们对我的每次努力都给予了赞赏与鼓励。如同以往，希望这些对各位有所帮助。

谢谢我的众多学生，在我尝试变换角度来解释某个概念的时候，他们能提出疑问（以及一脸迷惑）。希望本书的新版可以解除剩下的难题。

当然，最诚挚的感谢特别留给与我共同创作的作者，Randal。你给予我高度自由，让我可以在课堂上（以及书中）尝试各种讲述方法，而且时刻敦促我将这些阐述写入书中。还有一点务必要和Randal说：我被你深深感动，你热心劝勉他人，免于为了像你一样的官司而耗费大量的时间与精力，你是良好的典范。

谢谢我的妻子Jenna，谢谢你如此温柔体贴，为生活中大大小小的事感谢你。

来自brian

我必须先谢谢Randal，因为我就是从本书的第一版开始学习Perl的。而在1998年他要我进入Stonehenge开始讲课时，我又得再仔细读一遍！学好一件事的最好办法就是教别人学。在那之后，只要他认为我该学的，Randal都会指点我，不管是Perl还是其他方面的事，比如有次网络会议上，他决定我们应该用Smalltalk来展示，不要用Perl。我总是很惊讶于他渊博的知识。一开始就是他建议我写与Perl有关的东西。而现在，我也开始协助编写本书了。谢谢你Randal，能参与此事我感到非常荣幸。

在任职于Stonehenge的期间，跟Tom Phoneix见面的时间恐怕还不到两星期，但我多年来都是用他的教材上我们的“Learning Perl”课程。他的版本后来成为本书的第三版。在使用他的教材时，我也学到了解释某些概念的新方式，也深入了Perl的更多领域。

说服Randal让我参与“小骆驼书”的改版之后，我负责写企划书、维护全书大纲以及版本控制。我们的编辑Allison Randal不但在这些事情上给予了很多帮助，在收到我发出的大量邮件后也毫无怨言。在Allison转向其他工作后，我们的新任编辑Simon St.Laurent也极其负责地扮演着编辑和O'Reilly公司内部人员的双重角色，相当有耐性地一直等到月相宜人之时，才开始向我提出新的修改意见。

来自我们大家

感谢所有审校人员，谢谢David H.Adler、Alan Haggai Alavi、Andy Armstrong、Dave Cross、Chris Devers、Paul Fenwick、Stephen B. Jenkins、Matthew Musgrove、Jacinta Richardson、Steve Peters、Peter Scott、Wil Wheaton和Karl Williamson。感谢你们对本书草稿所提出的宝贵意见和建议。

感谢我们的众多学生，这些年来让我们知道这个课程的哪些内容需要调整改善。正是因为你们，我们今天才得以对本书如此自豪。

感谢诸多Perl推广组（Perl Mongers）在我们访问各位的城市时给我们宾至如归的招待。期待着与你们再次相见。

最后，向我们的朋友Larry Wall送上最诚挚的谢意，感谢你与大家慷慨分享这个新颖又强大的工具（也是玩具），让我们能够更快、更简单并且更有趣地完成工作。

第一章

简介

大家网
TopSage.com

欢迎阅读这本“小骆驼书（Llama book）”！

自1993年以来，本书（第六版）已经拥有超过50万名的读者。我们[注1]写这本书的时候非常开心，所以最起码的，希望那些广大读者们也能够喜欢这本书。

问题与答案

你可能会有一些关于Perl的问题，并且在快速浏览过本书后，还可能会提出一些有关本书的问题。所以，我们打算先用第一章予以回答。至于那些我们没有解答的问题，我们会告诉你如何获取答案。

这本书适合你吗？

如果你的个性和我们差不多，我想你一定有过这样的经历，书一定要买到手，才能撕开薄膜纸翻阅，所以这个问题好像没什么意义。在我们完成本书新版时，Borders书店已经关掉了许多店面，其他图书零售商也好不到哪里去。但也许你正在阅读的是下载来的电子版，或者是Safari Books Online上的网页版，这还好些。要是书都不让人翻，怎么知道它合不合适呢？又怎么让我们用这里的话提醒读者呢？

注1：　明确地说，本书第一版的作者是Randal L. Schwartz，第二版是Randal L. Schwartz与Tom Christiansen合著，第三版是Randal与Tom Phoenix合著，而现在是Randal、Tom Phoenix与brain d foy三个人一起合著。因此，本书提及的“我们”指的是最后那三位。现在，如果你怀疑我们如何能在卷首就说“写这本书的时候非常开心”，答案其实很简单：我们写书的方式，是从最后章节开始，然后按照各自的节奏由后往前推进。尽管听起来很奇怪，但老实说，当我们写完索引之后，剩下来的部分就完全不成问题了。

这不是一本参考书。这只是一本非常初级的教授Perl基础的教材，用里面的知识写点自己用的小程序应该不在话下。每一个主题都不会深入到所有细节，有些概念会贯穿在几个章节中，结合相关内容做进一步介绍。

我们希望读者至少能有一点基本的编程概念，并且确实是出于实际需要来学Perl语言的。你应该至少用过命令行终端，编辑过文件，运行过一些哪怕不是用Perl写的程序。你也应该知道变量、子程序等相关概念，而你现在要做的就是看看Perl是如何做的。

但这也绝不是说如果你从未接触过终端程序，或者从来没写过一行代码，就一定会茫然无措。第一次阅读一定会有些不太理解的东西，不过慢慢地你就会联系起来融会贯通。用过这本书的初学者一般只会碰到些小磕小绊的麻烦，其实关键在于不要一开始就担心那些不太明白的东西，只管专注于我们向你阐述的内容，以后你慢慢地就会明白。要成为经验丰富的程序员还有很长的路要走，迈出第一步才是至关重要的。

而且，这不应该是你读过的唯一一本Perl图书。本书仅仅是一份入门材料，无法包罗万象。本书的目的是帮助你跨出第一步，走对路。接下来再读我们的其他书籍，比如《Intermediate Perl》（在写本书时，第二版即将出版）以及更为高级的《Mastering Perl》。当然，完整而全面的Perl参考还非《Programming Perl》莫属，江湖人称“大骆驼书（Camel book）”。

另外需要明确的一点是，虽然本书谈及最新的Perl 5.14，但对老版本来讲，绝大多数内容都是适用的。如果你用的是Perl之前的版本，也丝毫不会影响学习Perl的基础概念。我们所涵盖的最低版本是Perl 5.8，即便如此这也是十年前就已发布的版本，所以完全不必担心。

为何有这么多的脚注？

感谢你注意到了，本书确实有很多脚注。如果你觉得讨厌，直接忽略好了。因为Perl语言中到处都充斥着例外，所以最好的方式就是补充脚注说明。应该说这是好事，现实生活中不也到处充满例外么。

就是因为有这些例外，我们不能昧着良心说完“fizzbin操作符可用来对hoozistaitc变量进行frobnicate处理”后却不加上脚注说明例外情况[注2]。我们向来严谨行事忠于事实，所以无法容忍省略脚注不表。不过要是你略过这些脚注不看，多少也可以算是忠于

注2：　除了当你趁星期二停电的机会玩fizzbin的时候（译注：美剧《星际迷航》中，fizzbin是其中一个扑克游戏，有很多奇怪的例外规则，比如周二不能翻开第二张牌）；或者是在5.12版之前的 Perl中，在某个定义了原型的子程序中的循环块内使用use integer编译命令。

事实吧（这么说逻辑上确实成立，很有意思）。一般来说，脚注都是些额外的并不涉及核心知识的说明。

许多例外与可移植性有关。Perl来自Unix系统，而且目前仍扎根于Unix中，与Unix息息相关。但无论是否因为在Unix以外的系统上运行造成的（或是别的什么原因），我们都会尽力呈现可能出现的意外状况。我们希望不懂Unix的读者们也能认为这是一本相当好的Perl入门书（而且你也可以因此而免费学到一点Unix知识）。

其他例外状况则与所谓的“80/20”定律有关。也就是说，Perl里面80%的功能可以用文档中20%的部分加以描述，而另外20%的功能却需要占据其他80%的篇幅。所以，为了保持本书的篇幅短小，我们在正文中介绍那些简单明了的东西，把深入高级的部分留在脚注中介绍（脚注采用小一些的字体，可以说明更多内容）[注3]。不看脚注读完本书后，你可以花点时间回过头来读一下脚注。参考也好，好奇也罢，这些脚注还是非常有趣的，有许多好玩的计算机笑话。

关于习题和解答？

每章结尾都备有若干习题，因为我们三人用这份教材教过上千名学生[注4]，实践证明效果不错。所以我们精心设计了这些习题，让你有机会体验一下多数人容易犯的错误。

不是说我们希望你犯错，而是你需要这样的机会。大部分错误在你的Perl编程生涯中迟早都会出现，所以不如提前经历一下，好有所准备。一旦有过前车之鉴，那么届时完成实际任务时，就算赶进度也不会再犯相同的错误了。如果做习题时碰到困难，也不必担心，随时都可以查阅附录A里的说明，我们会对习题做示范解答，并提示一些相关的内容，比如其中易犯的错误等。当你做完习题后，可以到这里核对一下答案。

在你努力尝试解决问题前，请尽量不要偷看答案。通过自己探寻答案而完成的习题，学习效果要比直接看答案好得多。就算一直想不出头绪来，也不必用头撞墙，先跳过它，翻到下一章好了，没关系的。

即便你没犯任何错误，在做完习题后也应该看一下解答。有些细节你可能未曾注意，看看解说或许能让你眼睛一亮。

想要额外练习的话，可以翻翻《Learning Perl Student Workbook》这本书，它针对每个章节都增补了许多习题。

注3：　我们甚至讨论过把整本书做成脚注以节省页数，但是“脚注的脚注”听起来有点怪。

注4：　当然不是一次教那么多。

习题前标的数字是什么意思？

每个习题之前都会有个数字，是以方括号框起来的，看起来像这样：

1. [2]当方括号里的数字2出现在题目前面时，表示什么意思？

这个数字是我们（非常粗略地）估计你完成这部分练习需要花费的时间。那只是非常粗略的估算，所以如果你已经全部完成（包括编写、测试和调试）却只用了一半的时间，或者花了两倍时间还没完成，都不要太惊讶。不过就算你真的被难倒了，我们也不会告诉别人，你的答案是偷看附录A得来的。

如果我是Perl讲师？

如果你想在自己的课程里使用本书作为教材（历年来都有不少人这么做），请留意我们对各章习题的设计，尽量让大部分学生在45分钟到1小时内完成，再留下一些休息时间。某些章节的习题需要的时间会少一些，某些章节则要多一些。之所以会出现这种情况，是因为填完方括号里的数字后，我们才发现自己竟然不太会做加法（还好我们知道怎么让计算机帮我们做这件事）。

之前提到过，我们还有一本辅导用书《Learning Perl Student Workbook》，它针对每个章节都额外增加了若干习题。如果你有这本工具书的第四版的话，请注意调整一下章节顺序，这次新版我们新增了一章，另外还调整了部分章节的先后次序。

“Perl”这个词表示什么意思？

Perl有时候被称为“实用摘录与报表语言（Practical Extraction and Report Language）”，但也会被称作“病态折中式垃圾列表器（Pathologically Eclectic Rubbish Lister）”。除此之外，这个词的缩写还可以展开为其他不同的名称来诠释。Perl是个溯写字（backronym），而不是缩写词（acronym），这是因为Larry Wall，也就是Perl的缔造者，是先想出要用这个词，然后再考虑如何展开解释的。要争论哪种全名才是正确的并无太大意义，无论哪种Larry都认可。

你可能会在某些技术文章里看到以小写p来表示“perl”。一般说来，大写P表示的“Perl”指的是程序语言，而小写p表示的“perl”指的是实际编译并运行程序的解释器。按照惯例，在表示命令行程序时我们会以这样的格式书写：*perl*。

Larry为什么要创造Perl？

20世纪80年代中期，Larry想要为类似新闻组的文件体系写一个bug（缺陷）汇报系统，

当时用的是*awk*，但马上发现*awk*无法满足他的需求。于是，作为一名以懒惰为美德的程序员[注5]，Larry决定从根本上解决这类问题，写一个通用的多用途工具，让它不仅能解决眼下这个问题，将来也能在别的地方派上用场。于是，Perl第零版就这样诞生了。

Larry干吗不用其他语言？

世界上不缺乏程序语言，不是吗？但在当时，Larry却找不到任何一种真正符合他需要的语言。如果时下某种语言在当年就能够出现的话，Larry或许就会直接用它了。他当时需要的是像shell或*awk*一样能快速编程，又具有类似*grep*、*cut*、*sort*、*sed*等高级工具的功能[注6]，而不必回头使用像C这种类型的语言。

Perl试图填补低级语言（如C、C++或汇编语言）和高级语言（如shell编程）之间的空白。低级语言通常既难写又丑陋，但是运行速度很快而且不受限制。不管在哪台机器上，要想赢过写得好的低级程序的运行速度，恐怕难于登天。它们几乎可以做所有工作。而高级语言则是另一个极端，它们通常速度缓慢、难写又丑陋，并且限制重重。如果系统上不提供执行某些必要功能的接口，那么shell程序会有很多工作无法完成。而Perl则相当容易，几乎不受限制，速度又很快，只是看起来有点别扭。

好吧，现在让我们来细数一下上面提到的Perl的四大特点。

首先，Perl很容易。不过接下来你马上会发现，其实这指的是容易使用。学习Perl并不简单。如果你会开车，你一定是花了好几个星期或几个月的时间来学习，最后开起来才会驾轻就熟。当你花在写Perl程序上的时间和学开车的时间一样长时，Perl对你而言就是很容易的东西了[注7]。

Perl几乎不受限制，几乎没什么事是Perl办不到的。你大概不会想用Perl来编写中断－微内核层次（interrupt-microkernel-level）的设备驱动程序（尽管已经有人这么做了），但一般人用来处理日常琐事的程序，从临时需要完成某项任务的小程序到企业级的大型应用程序，都很适合用Perl来写。

Perl的速度通常很快。这是因为所有Perl开发者同时也都是Perl用户，所以我们当然都希

注5：　我们说Larry懒惰，并不是说他的坏话，懒惰其实是一种美德。手推车是由懒得扛东西的人发明的，书写是由懒得记忆的人发明的，Perl的创造者也是懒人，若不发明一个新语言就懒得干活。

注6：　要是你不知道这些是什么东西，请别担心。重点在于它们是Larry当时手上能够用的Unix工具程序，但是功能不够强大。

注7：　当然我们并不希望你开车时碰到一样多的失灵。

望它能够运作如飞。假设有人为Perl加上某个很酷的功能，可同时会让其他程序变慢，那么Larry几乎一定会拒绝加入这项新功能，直到找出让它变快的解决办法为止。

Perl代码有点难看，这倒是事实。Perl的标志是骆驼，这来自于值得尊敬的“大骆驼书”（即《Perl语言编程》）的封面，这本“小骆驼书”（以及另一本姐妹书，“羊驼书”）算是该书的表亲。骆驼长得也有点丑陋，但它们努力工作，哪怕在严酷环境下也一样不辞辛劳。骆驼能在种种不利的条件下帮你把事情搞定，尽管它长相丑陋，气味难闻，偶尔冷不丁还会对你吐上几口口水。怎么说呢，Perl有时候确实有点像它。

Perl算容易，还是算难？

它简单好用，但确实不太好学。当然，这只是一般而言。在Larry设计Perl时，他必须做出许多权衡取舍。每当有机会可以让程序员用起来无比痛快但让初学Perl的人觉得难以理解的时候，他几乎总是站在程序员这边。原因很简单，学只学一次，用却是一辈子可以用下去的[注8]。Perl有不少简便操作的写法，可以让程序员节省大量时间。比如大部分函数都具有默认行为，而这种默认行为也是绝大多数人在使用该函数时想要采取的操作。所以，像下面这样的Perl代码其实随处可见[注9]：

```
while (<>){
    chomp;
    print join("\t", (split/:/)[0,2,1,5]), "\n";
}
```

要是不用Perl的默认行为与简写，那么上面这段代码的大小可能会增长十几倍，这么一来，阅读与编写的时间也会大幅增加。并且需要用到更多变量，从而使得维护和调试也复杂一些。如果你已经能看懂一点Perl，你会发现上面的代码里其实根本就没有变量，注意，这才是问题的关键。实际上这里用到的变量都是以默认行为来工作的。为了将来写起来方便流畅，我们得先投入一点时间，学习一下默认变量的使用规则。

其实，简写或者缩写来源于现实生活，比如大家在英语里经常会看到缩写，从来不会觉得有什么不妥。是的，“willnot”跟“won’t”这两种写法的本质意义一模一样。但大家都会说“won’t”，而不太会说“willnot”。一来因为比较省时间，二来大家都熟悉这

注8：　如果你每周或每个月只花几分钟的时间在程序设计上，容易学习的语言会比较合适，因为下次使用时，你可能就忘光了。Perl是为每天至少花20分钟写程序（并且是以Perl程序为主）的程序员设计的。

注9：　在这里我们无法详细解释所有细节，大体上，这段程序会从文件读入一些数据，并把数据从原来的格式转成另一种格式。不必担心，程序里用到的所有功能，本书都将会逐一介绍。

样的模式。同样，Perl也会缩写常用的"字句"，把许多难写的程序浓缩成简洁有力的"成语"，好让维护人员能够快速地"听说"Perl。

一旦熟悉Perl之后，你就可以花更少时间去摆弄shell的引用（或C语言的声明），有更多的时间来浏览网站。这是因为Perl能让你事半功倍。Perl简明的语法让你能够（毫不费力地）建立很酷并且流畅自然的解决方案，或是用途广泛的工具程序。由于Perl既跨平台又随处可用，所以这次实现的工具可以在下次的任务里沿用，节约出来的时间让你多做些其他有益的事情，不是很好么。

Perl是非常高端的语言。这表示Perl代码的密度和信息量也相当高，Perl程序的长度大约是等效C程序的30%到70%左右。随之而来，编写、阅读、调试和维护Perl程序的效率也非常高。哪怕只写过一点程序的人都明白，当子程序小到能够放进一个屏幕时，编写时就不用上下滚动来回查看。此外，既然程序里的bug数量大致与源代码长度成正比[注10]（而不是与程序的功能成正比），那么较短的Perl程序代码平均起来含有bug的数量也会少很多。

像其他任何一种语言一样，Perl也能写出叫人看不懂的程序，就好比它是"只写的（write-only）"。但只要你稍加用心，就可以避免这项常见的恶名。没错，Perl程序对门外汉来说，看起来可能像CPAN上的线路噪声，但对经验丰富的Perl程序员来说，它就像大交响乐团的总谱。你只要遵照书里的指引，就能写出易于阅读且易于维护的程序，当然，用这些程序是赢不了戏玩Perl代码大赛（Obfuscated Perl Contest）的。

Perl怎么会这么流行？

Larry稍加测试Perl，并在各处略作改进后，就把它发布到Usenet的读者社群，也就是一般所谓的"网络（Net）"。这群散居世界各处的（上万名）用户给了Larry许多及时反馈，希望Perl做这做那，而其中有许多都是Larry从来没想过要用他的Perl去处理的。

可结果是，Perl不断成长。它的功能变多了，能运行它的平台也增加了。当年这个只能在少数几种Unix系统上运行的小语言，而今长成了具有上千页在线自由文档、成打书籍、数个主流Usenet新闻组（以及成堆的新闻组与邮件列表）、新闻组里无数读者，并且时下近乎所有系统皆可使用的版本。当然，其中也包括这本"小骆驼书"。

现在的Perl发展得怎么样了？

Larry Wall已经不再亲自编写所有Perl核心代码，但他仍然会指引开发方向并作关键性抉择。目前维护Perl的是一个热心的开发者团队，我们称之为"Perl 5掌门人（Perl 5

注10： 要是程序里有任何一段代码的篇幅超过屏幕显示范围，那么bug数目还会显著上升。

Porters）”。可以加入他们的邮件列表<*perl5-porters@perl.org*>参与讨论或关注最新动态。

在我们写下这段文字的同时（2011年3月），有许多事情正围绕着Perl发生。在过去几年里，有许许多多的人投入到下个重要版本Perl 6的开发中。

简单来讲，Perl 6是一门完全崭新的语言，甚至连目前初步的实现都已经改名为Rakudo。Perl 6是在2000年的时候发起的，原来打算要取代Perl 5，但随后的开发时起时落，进展缓慢，一度处于低迷状态，而与此同时，Perl 5.6，5.8，甚至5.10都已陆续发布面世。历尽各种困难和分心之后，Perl 5的开发又再度活跃起来，而Perl 6渐渐冷了下来。听起来挺讽刺的，也许吧。

不过，近年来Perl 5的开发逐渐恢复，并且现在已经到了差不多每个月发布一个测试版，每年发布一个核心版的程度。本书之前一版仅仅讨论到5.10，没有机会覆盖随后发布的5.12，这多少有些令人抱憾。不过现在这个新版正好是Perl 5.14发布之时所作，还算欣慰，但Perl 5掌门人已经在捣鼓Perl 5.16了。

哪些事情最适合用Perl来做？

Perl很适合在三分钟内写出“虽然难看但是能用”的一次性程序，Perl也很适合用来编写用处广泛、需要十几个程序员花三年时间才能完成的大型程序。当然，你会发现大部分Perl程序从构思到完成测试，一般只要不到一个小时的时间。

Perl擅长处理整体来说“约有90%与文字处理有关，10%与其他事务有关”的问题。这似乎占了当前编程任务需求的绝大部分。在理想的世界里，所有程序员都会每一种语言，进行任何项目时他们都能选择最适合的语言。但绝大多数时候他们会选择Perl[注11]。在Larry创造Perl的那个年代，有关Web的概念甚至还没从Tim Berners-Lee的头脑里蹦出来，但这两者却在后来通过网络走到了一起。有人说Perl在20世纪90年代初期的扩张，让许多信息能迅速转换成HTML格式的文档，从而推动了互联网的蓬勃发展。是的，Perl确实是编写小型CGI脚本（Web服务器调用的程序）的最佳语言。直到现在，还有许多搞不清楚状况的人会提出像“CGI不就是Perl吗？”或者“除了CGI之外，Perl还有什么用呢？”这样的问题。老实讲，很搞笑。

注11：不要只听我们的一面之词。如果想知道Perl和X语言哪个比较好，最好的办法就是把两者都学会，看看到后来你最常用的是哪个。用得最多最顺手的，自然就是最适合你的。不管怎样，从结果上来讲，学过 X 语言之后加深了你对Perl的理解（或者反过来），那也不失为一种收获，所以并不是浪费时间。

哪些事情不适合用Perl来做？

那么，既然Perl能做的事情这么多，哪些事不适合用它来做呢？好吧，如果你想做出封闭式二进制可执行文件（*opaque binary*），请不要使用Perl。所谓的“封闭式”，指的是取得或购得你程序的人无法从程序里看到你的秘密算法，因此也无法协助你维护或调试。当你把Perl程序给某人时，通常给的是源代码，而非封闭式二进制可执行文件。

不过，要是你真的想要封闭式二进制可执行文件的话，我们必须遗憾地告诉你，其实并没有这种东西。只要有人能安装并运行你的程序，他就能将它还原成各种程序语言的源代码。当然，通过这种方法取得的代码多半和你原来的不同，但它毕竟是某种源代码。不幸的是，要保护你的秘密算法，真正的办法只有一种：聘用足够多的律师。他们能写出一份授权条款，然后声明“你可以用这个程序做这件事，但是不能做那件事。要是你违反了我们的规定，我们有足够多的律师会叫你后悔莫及。”

如何取得Perl？

你的机器上可能已经有Perl了。至少，在我们接触过的机器上都找到过Perl。许多系统都附带Perl，大多都已预先安装好，也有需要自己安装的。系统管理员一般都习惯在他管理的每台服务器上安装Perl。不过话说回来，就算你的系统没有提供Perl安装包，你总还是可以从别处免费下载得到。一般各种Linux或者*BSD系统，包括Mac OS X等，都会预装Perl。另外还有些公司会提供第三方的Perl版本，比如ActiveState（*http://www.activestate.com*）就为某些平台，包括Windows，提供预先编译好的版本。另外，你还可以下载Strawberry Perlfor Windows（*http://www.strawberryperl.com*），除了标准的Perl核心代码模块之外，还会附带用于编译与安装第三方Perl模块的工具。

Perl有两种不同的授权条款。对只是使用Perl的大部分人来说，这两种条款并没有什么差别。不过如果你想修改Perl，请详细阅读这两份授权条款，上面对发布变动过的Perl有些限制。对于不想修改Perl的人而言，授权条款基本上是说：“这是自由软件——你爱怎么用都行。”

事实上，它不仅是自由软件，还能在几乎所有自称为Unix、具有C编译器的系统上顺利运行。你只要下载它，键入一两条命令，Perl就能自行配置与安装。还有更好的办法，就是找到系统管理员，让他键入这一两条命令来帮你安装[注12]。除了Unix和类似Unix的系统之外，对Perl上瘾的人还将它移植到了其他平台上，比如Mac OS X、VMS、OS/2，

注12：　如果系统管理员不能安装软件，要他们做什么？如果难以说服管理员安装Perl，可以用买比萨请客作为交换条件。我们还没碰过能拒绝免费比萨（或其他容易弄到的食物）的系统管理员。

甚至还包括MS/DOS以及所有Windows的现代版本。在你看到这句话时，可能又增加了更多支持平台[注13]。这些Perl的移植版本（*ports*）通常都会附带安装程序，有些甚至比Unix的安装程序更容易使用。具体信息请参考CPAN网站上“ports”部分中的链接（译注：*http://www.cpan.org/ports/index.html*）。

CPAN是什么？

*CPAN*就是Perl综合典藏网（Comprehensive Perl Archive Network），可以说是非常方便的Perl一站式大卖场。里面有Perl本身的源代码、各种非Unix系统的安装程序[注14]、范例程序、说明文档、扩展模块以及跟Perl相关的历史邮件存档。简单来说，CPAN无所不包。

CPAN有数百个镜像站点分布在世界各地，请从*http://search.cpan.org/*开始，慢慢浏览整个典藏网吧。如果你无法访问网络，也可以到附近卖技术书籍的书店找找，可能会有CPAN部分实用内容构成的CD或DVD出售。不过，请确认它是最新版本。既然CPAN每时每刻都在更新，那么两年前的库存就已经算是老古董了。更好的办法就是找个可以访问网络的朋友，帮你将当天的CPAN刻录成光盘。

如何得到Perl的技术支持？

好吧，既然你手上已经有了完整的源代码，那么何不自己动手修掉bug呢？

你不禁要皱眉了，对吧？不过从某种程度上说，这确实是件好事。因为Perl没有“源代码保护条款”，所以理论上任何人都可以修正其中的bug。不过一般而言，当你找到并确认某个bug时，多半已经有人将它修好了。Perl是由全世界数千人共同维护的。

当然，我们并不是说Perl有一大堆bug，但是它充其量只是一个程序，而所有程序必定至少有一个bug。拥有Perl源代码到底有什么好处呢？我们不妨想象一下，假设你现在用的是一种叫做Forehead的语言，而授权出售它的公司又很大，老板是某个发型难看的亿万富翁（纯属虚构，大家都知道没有Forehead这种语言）。如果你发现Forehead里有个bug，你能做什么呢？首先，你可以汇报问题。然后，你可以怀抱希望——希望他们会尽快修复这个问题，希望下个版本卖得不要太贵。你还可以希望下个版本不要加上新的

注13：并非完全如此，在我们写此书时，在Blackberry上还不能这么做——这个系统实在是太过繁复，即便是简化后的系统也是如此。但已有传闻说它可以在Win CE上使用。

注14：一般来讲，在Unix系统上自己动手编译Perl源代码才是最佳选择。考虑到其他系统也许没有C编译器，或者缺少用于编译的工具，所以CPAN也一并准备了预先编译好的二进制可执行文件，下载下来就能使用。

功能和新的bug，并且希望该公司不会因为犯了反托拉斯法而被拆成两半。

但假如是用Perl，你掌握着所有源代码。就算在某种极端情况下你无法修复某个bug，你总还可以花钱雇用某个程序员或开发团队来帮你解决。要是你买了台新式计算机，Perl还不能在上面运行，你也可以自己进行移植。如果你想要某个新功能，你也知道该怎么办。

还有别的技术支持方式吗？

当然了！我们最喜欢的技术支持之一就是Perl推广组（Perl Mongers）。它是全世界的Perl用户组织，具体信息可以到*http://www.pm.org/*上查阅。在你居住的城市附近就应该有个分部，可以找到专家或认识专家的人。要是附近没有，不如你自己成立一个吧。

当然，作为第一手的技术支持，不要忘了你手边还有Perl说明文档可供翻阅。除了随Perl安装的核心文档，你还可以到CPAN（*http://www.cpan.org*）上去看看，也可以到其他类似的站点——比如*http://perldoc.perl.org*就提供了Perl核心文档的HTML版本和PDF版本，还有*http://faq.perl.org/*上有最新版的perlfaq，即Perl常见问题汇编。

另一个比较权威性的参考是《Perl语言编程》这本书（译注：已经发行的最新版是第三版，其简体中文版已由中国电力出版社于2001年出版发行。目前即将面市的第四版将于2011年11月出版发行，其中文版翻译也在计划之中。），因为该书封面动物的关系，它通常被称为“大骆驼书”（就像本书被叫成“小骆驼书”一样）。大骆驼书里包含了完整的参考信息、一些教学范例以及许多与Perl相关的杂项信息。另外还有一本口袋大小的《Perl 5 Pocket Reference》，作者是Johan Vromans（O'Reilly出版），很适合拿在手上阅读（或是放进口袋里）。

如果你想找人问问题，Usenet上有许多新闻组，此外还有很多邮件列表可供咨询[注15]。无论何时，在某个时区都会有专家回答Usenet上Perl新闻组里的问题，Perl是个日不落帝国，这表示在你提出问题的几分钟后通常就会有人提供解答。但如果你事先没查过Perl自带的说明文档和常见问题汇编的话，几分钟之内就会挨骂。

Perl官方的Usenet新闻组位于*comp.lang.perl.**这一级。在编写此书时，它一共有五个组，但也会随时间的改变而有所增减。你（或是在你的系统上负责管理Perl的人）通常要订阅*comp.lang.perl.announce*组，它是个低流量、仅仅只有重要公告信息的组，也包括了Perl在信息安全方面的公告信息。如果需要人指点Usenet的使用方式，请就近询问专家。

注15：　比较完整的邮件列表列在*http://lists.perl.org*上。

此外，也有一些从事Perl讨论的社群网站。其中很受欢迎的The Perl Monastery（*http://www.perlmonks.org*）上有许多Perl书籍和专栏的作家，至少包括了本书的两位作者。你也可以看看近年来比较热门的Stack Overflow（*http://www.stackoverflow.com*）站点，上面有各式各样有关Perl的问答和讨论。

你也可以到*http://learn.perl.org/*以及对应的邮件列表<*beginners@perl.org*>上咨询。许多著名的Perl程序员也会开设技术类博客，时不时地发布一些有关Perl主题的文章，其中绝大部分都可以到聚合站点Perlsphere（*http://perlsphere.net/*）上阅读。

如果你需要签一份Perl的技术支持服务合约，有好几家公司会愿意收你的钱。不过通常情况下，那些免费的支持资源就已足够。

如果发现Perl有bug，我该怎么办？

当你发现Perl有bug时，请务必再次查阅Perl文档[注16]一遍[注17]。Perl有许多有趣的特性、别出心裁的用法，所以很多时候人们会把某些特性或者另类的用法误以为是bug。另外，请确定当前安装的是最新版的Perl，老版本的bug多半已经在新版中清除干净。

如果你99%确定自己找到了真正的bug，请再问问看周围的朋友、公司的同事，或者在参加当地Perl推广组活动或Perl集会时提问。它很可能仍旧是某项特性，而非bug。

要是你100%确定找到了一个真正的bug的话，写一个针对它的测试脚本。（什么？难道你从没写过测试么？）一个写得好的测试应该是任何一个Perl用户都能运行并重现你所找到的问题的。有了能清楚重现bug的测试案例后，可以用工具程序*perlbug*（随Perl一起发布）报告此bug。一般是通过发送邮件的方式报告给Perl开发者团队，所以请务必在准备好测试程序后再用*perlbug*报告，否则他们难以为继。

如果一切顺利，报告发送后不出几分钟，你就会收到来自开发团队的回应。如果问题已经解决，他们会给你发送一个补丁文件，然后你可以自己打上补丁，继续手头的工作。当然，最坏的情况下也可能完全没有回应，Perl开发者并不是非得要立即处理收到的bug报告，严格来说他们没有这样的义务。不过既然我们大家都这么热爱Perl语言，谁也不会坐看bug在自己眼皮底下逍遥法外的。

注16：　即便是Larry自己，也时常查阅文档，确认一些信息。

注17：　有时还得反复查上两三次。我们往往在浏览文档中某个异常状况的线索时发现新的细节，而这些细节最后都成了讲座或专栏文章里面的有趣花絮。

我该怎么编写Perl程序？

差不多是问这个问题的时候了（即使你还没问）。Perl程序只是一个纯文本文件，你可以用任何一个自己喜欢的文本编辑器来创建或编辑Perl程序。Perl不需要特殊的开发环境，虽然也有厂商提供专门的商业软件。我们从未用过这类软件，所以没有发言权，无法向你推荐（相反，我们倒是有一堆理由可以说明根本没必要用这类商业软件）。不过，你自己的开发环境归根结底还是你自己说了算。随便问三个程序员他们用什么开发环境写程序，或许会得到八种不同答案。

一般来说，你应该使用程序员专用的编辑器，而不是普通的编辑软件。两者有什么不同呢？程序员专用的编辑器可以快速方便地执行写程序时常用的那些操作，比如调整一段代码的缩排，或者查找并定位成对的括号。在Unix系统上最受欢迎的两种程序员专用编辑器是*emacs*和*vi*（以及它们的衍生版本），而BBEdit和TextMate则是Mac OS X系统上深受欢迎的文本编辑器。在Windows上有很多人推荐使用UltraEdit和PFE（Programmer's Favorite Editor）。至于其他可供选用的文本编辑器，不妨看看列在*perlfaq3*文档中的清单。如果不知道自己的系统上该用哪个文本编辑器，请咨询附近的系统管理员或者专家。

本书习题需要编写简单的程序，但长度都不超过二三十行，所以你用哪种文本编辑器都没问题。

有些初学者会尝试使用文字处理器来代替文本编辑器。我们不建议这种做法，首先写起来不方便，其次还有可能根本无法运行写出来的代码。我们不会强制你作何选择，不过至少请确认文字处理器以“纯文本”格式保存你的文件，以免采用默认保存格式而导致无法运行程序。几乎所有的文字处理器都会提醒你，正在编辑的Perl程序有一大堆拼写错误，你还应该少用一点分号。

在某些情况下，你可能需要在一台机器上写程序，再传送到另一台机器上运行。这个时候，请使用“文本模式（text mode）”或者“ASCII模式（ASCII mode）”来传输程序。记住，千万不能是“二进制模式（binary mode）”。即便是文本文件，不同系统对待换行符的方式也有所不同，所以碰到无法理解的换行符时，某些老旧的Perl还可能会中断运行。

一个简单的程序

按历来习惯，任何一门根植于Unix文化的程序语言入门书都会以“Hello,World”这个程序作为开头。所以，下面是它在Perl里面的写法：

```
#!/usr/bin/perl
```

```
print "Hello, world!\n";
```

假设我们已经将上面两行代码输入到文本编辑器里了（暂且先别管程序各部分是什么意思，我们很快就会讨论到）。一般来说，程序可以以任何文件名保存。Perl程序并不需要用什么特殊的文件名或扩展名命名，甚至能不用扩展名就最好不要用[注18]。不过，某些Unix以外的系统上也许必须使用*.plx*（代表PerL eXecutable）之类的扩展名，进一步的信息请参考你的系统中Perl发行版本的相关说明。

接下来，你可能还需要告诉系统，该文件是一个可执行程序（也就是一个命令）。不同的系统需要的操作方式也会有所不同，也许只需要将程序文件存储到某个地方就行了（多数时候在你当前的工作目录也行）。在Unix系统上，你可以使用*chmod*命令将程序文件的属性修改为可执行的，如下所示：

```
$ chmod a+x my_program
```

其中，最前面的美元符号（以及空白）代表命令行提示符（shell prompt），在你的系统上多半会不太一样。如果执行*chmod*时习惯用755之类的数值而不是a+x这样的符号来代表权限，那当然也没有问题。不管用哪种写法，它都能告诉系统，这个文件现在已经是一个可以执行的程序了。

现在，已经可以运行这个程序了：

```
$ ./my_program
```

命令开头的点号与斜线表示要在当前工作目录（current working directory）里查找这个程序。并不一定每次都需要这么写，但在完全了解它们的意义之前，每次执行命令时加上它总归不会有错的[注19]。如果写完第一次运行就能如期运行，那简直就是奇迹。一般多少会有些低级错误导致的bug，简单修整一下，再运行一次就好了。不必每次运行前都用*chmod*，因为一旦设定文件的可执行属性后，这个状态会一直保留，不会发生变

注18：为什么不带扩展名比较好？假设你写了一个计算保龄球分数的程序，然后告诉所有朋友它的文件名是*bowling.plx*。之后某一天，你决定以C语言来改写它。要保留原来的文件名，继续让人以为它是用Perl写的吗？还是要跟大家说它换了个新名字呢？（拜托，可执行文件的名称请不要叫*bowling.c*！）正确答案是：程序是用什么语言写的跟用户一点关系也没有。所以，程序最初的文件名就应该叫*bowling*。

注19：简而言之，这是为了防止shell执行同名的其他程序（或shell的内置命令）。初学者常犯的一个错误，就是把第一个程序取名为*test*。很多系统已经有同名程序（或shell的内置命令）了，所以不加路径直接运行的很有可能就是系统自带的程序或命令，而不是你写的那个。

化。（当然，如果因为运行*chmod*时没有赋予程序文件可执行权限，那么运行该程序时会报告“无此权限（permission denied）”这样的错误信息。）

这个简单的程序在Perl 5.10及其后续版本里还可以有另外一种不同的写法。这次不用`print`，改用`say`，它的效果基本相同，但却不需要输入换行符，并且减少了键入次数。由于这是个新特性，而你可能还没安装Perl 5.10，所以使用`use 5.010`语句告诉Perl，我们需要引入该版本中的新特性：

```
#!/usr/bin/perl
use 5.010;
say "HelloWorld!";
```

这个程序只能在Perl 5.10及其后续版本中运行，在本书后续章节中介绍到Perl 5.10的新特性时，我们都会明确告诉你这一点，并且使用`use 5.010`语句作为提醒。因为Perl总是将次版本号看作是三位数表示的，所以前导零千万不要省略，记住要写成`use 5.010`而不是`use 5.10`（Perl会把它当作`5.100`，一个目前我们还没发布的版本！）。

一般来说，我们只需要声明新特性初次引入的版本号即可。不过本书会谈论最新的Perl5.14版，所以有很多新特性是在该版本后引入的，这些特性的示例代码都应该添加下面这行：

```
use 5.014;
```

程序里写的是什么？

和其他“形式自由（free-form）”的语言一样，Perl通常可以随意加上空白符（像空格、制表符与换行符等），使程序代码更易阅读。不过多数Perl程序都会选择使用比较统一的格式标准，本书也不例外[注20]。我们强烈建议并鼓励使用适当的缩排，因为它能有效增加程序的可读性。缩排工作并不麻烦，许多专业的文本编辑器都能帮你自动处理。此外，好的注释也能改善程序的可读性，从而快速直观地理解程序要做的事。Perl里的注释是从井号（#）开始到行尾结束的部分。（Perl没有“注释块”的概念[注21]）。我们不会在本书展示的程序中使用很多注释，因为代码前后的正文本身就会对它做详细解释。但在你的程序里，请尽量加上适当的注释。

如果不管格式，仍然以之前的“Hello,World”来说，写成下面这样虽然也能运行，但老实说，这种写法实在是太诡异了：

注20：　在*perlstyle*文档中介绍了一些通用的缩排建议（但不是硬性规定哦）。

注21：　但也有许多模拟方法，具体技巧请参考文档的*perlfaq*部分中的说明。

```
#!/usr/bin/perl
    print#    这里的部分就是注释
"Hello, world!\n"
;    #不要把Perl代码写成这个样子!
```

第一行其实是个具有特殊意义的注释。在Unix系统里[注22]，如果文本文件开头的最前两个字符是#!，那么后面跟着的就是用来执行这个文件的程序路径。在上例中，该程序就是*/usr/bin/perl*。

事实上，Perl程序里最缺乏可移植性的就是#!那行了，因为你必须确定在每台机器上*perl*解释器是放在什么路径下的。幸好多数情况下不外乎*/usr/bin/perl*和*/usr/local/bin/perl*这两种。如果不是，你就得找出系统上的*perl*解释器程序究竟是藏在哪个路径下，然后改用此路径。在Unix系统上，还可以写成下面这种样子，通过在shebang行上执行外部命令自动帮你定位*perl*解释器的路径：

```
#!/usr/bin/envperl
```

如果在所有可查找的目录下都找不到*perl*解释器，就近请教一下系统管理员或是使用同样系统的朋友吧。但请记住，这仅仅是定位所能找到的第一个*perl*解释器，而它未必就是你希望使用的那个。

在非Unix系统中，传统上第一行会写成#! perl（其实这也是有用的）。至少，它能让维护人员在着手修正程序时，马上知道这是个Perl程序。

要是#!行写错了，shell通常会给出错误信息。它的内容可能会出乎意料，如“File not found（找不到文件）”或“bad interpreter（错误的解释器）”。这并不是说shell找不到你要运行的程序，而是说找不到程序中指定的*/usr/bin/perl*。我们也很想把这个报错信息改得更清楚明确些，但它不是Perl发出的，发出抱怨的是shell。

另一个你可能会碰上的问题，就是你的系统完全不支持#!行。这样一来，你的shell（或者不管其他什么类似shell的东西）也许会直接尝试运行你的程序，出来的结果多半会叫人失望或是出人意料。要是你搞不清某些奇怪的错误信息在讲什么，可以到*perldiag*文档中查阅。

“主（main）”程序完全由普通的Perl语句（statement）组成（子程序里的除外，这个我们稍后再谈）。这和C或Java语言不同，Perl里头没有以“main”命名的例程（routine）。事实上，好多Perl程序可能根本不用子程序。

注22：当然，我们指的是比较近现代的系统。“sh-bang”读作“sheh-bang”，就像在“the whole shebang”中的读音一样。这种写法是在20世纪80年代中期开始用起来的，对Unix这样历史并不太悠久的系统而言，这已经算是相当早期的功能了。

另外，Perl程序并不需要变量声明的部分，这点和其他语言不同。如果过去你一直习惯声明所有变量，那现在可能会有点不安。不过，这种特性使得我们可以编写“虽然有点难看但马上就能运行”的Perl程序。如果程序的长度只有两行，却将其中一行耗费在声明变量上，似乎不太值得。如果真的想声明变量的话，那也是一件好事情，我们会在第四章里讨论怎么做。

Perl语言的大部分语句都是表达式后面紧接着一个分号[注23]。下面的语句我们已经看到过好几次了：

```
print "Hello, world!\n";
```

你大概已经猜到了，上面这行会输出Hello, world!。接在这两个词后面的是转义写法的\n，其含义对于熟悉C、C++和Java的用户来说应该不陌生：它就是换行符（newline character）。当它跟在某些字符串后面打印出来时，光标位置会从行末移到下一行的开头，这样结束运行后，shell的提示符就可以在新的一行开始，而不必跟在程序输出的信息后面，所以在写这样的程序时，最好在每一行输出的信息后面都加上换行符作为结尾。下一章，我们将会看到更多关于换行符的转义写法以及其他各式“反斜线转义”（backslash escape）的介绍。

我该如何编译Perl程序？

只需要直接运行它就可以了。只此一步，*perl*解释器能一次完成编译和运行这两个动作：

```
$ perl my_program
```

运行程序时，Perl内部的编译器会先载入整个源程序，将之转换成内部使用的*bytecode*，这是一种Perl在内部用来表示程序语法树的数据结构，然后交给Perl的bytecode引擎运行。所以，如果在第200行有个语法错误，那么在开始运行第二行[注24]代码之前，Perl就会报告这个错误。如果你的程序中有一个运行5 000次的循环，它只会被编译一次，然后每次循环都以最快的速度运行。除此之外，为了提高易读性，不论你用多少注释和空白，它们都不会影响运行时（runtime）的速度。你甚至可以使用完全由常量组成的算式，它的值只会在程序开始时被计算一次，哪怕它是在某个循环中也只被计算一次，然后在后续的执行中重用计算结果。

不用说，编译是要花时间的——所以如果只是为了要迅捷地完成某个简短任务而去运行一个冗长的包含其他各种任务代码的Perl程序，会显得效率低下，因为花在编译上的时

注23：　分号的目的是分隔不同的Perl语句，而不是断行。

注24：　除非第二行恰好是编译时（compile-time）执行的指令，比如BEGIN块或者use语句。

间可能会比运行时间还要长。不过Perl编译器的运行速度非常快，通常编译时间只占运行时间的极少部分而已。

不过，有个例外的状况，如果你编写的是CGI脚本，它有可能每分钟会被Web服务器调用运行成百上千次（这样的使用频率非常高。如果像Web上的大多数程序一样，只是每天运行成百上千次的话，就没什么好担心的了）。大多数这类程序的运行时间都很短，所以重复编译造成的时间消耗问题可能会比较明显。如果你碰上这种情况，就需要想办法将程序代码编译后让它驻留到内存中，好让后续的调用跳过编译，直接运行。Apache Web服务器（*http://perl.apache.org*）的mod_perl扩展模块或者类似CGI::Fast这样的Perl模块都可以解决这个问题。

要是把编译后的bytecode存储起来，能否节省编译时间？或者更进一步地说，能否将bytecode转换成另一种语言，比如C语言，然后再进行编译？好吧，其实以上两种做法在某些情况下都是可行的，不过老实说这么做并没什么好处，程序不会因此变得更易使用、维护、调试或安装，甚至（由于某些技术性因素）还会让程序运行得更慢。

走马观花

读到这里，你一定想看看可以派上实际用场的Perl程序到底是什么样子（要是你不想，麻烦暂时合作一下嘛）。请看：

```
#! /usr/bin/perl
@lines = `perldoc -u -fatan2`;
foreach (@lines) {
    s/\w<([^>]+)>/\U$1/g;
    print;
}
```

如果这是你第一次看到这样的Perl代码，可能会觉得有点怪异（事实上，每次读到这样的Perl代码，都会让你觉着挺奇怪的）。不过，我们会逐行讲解这段程序，看看它到底做了些什么（下面的介绍会非常简略，毕竟这一节只是“走马观花”。程序里用到的所有功能在后面的章节中都会详加说明，你不需要现在就把程序彻底搞懂）。

第一行是我们介绍过的#!行。在你自己的系统上，可能需要修改下，确认使用的是正确的路径，具体请参考前文所述。

第二行运行了一个外部命令，通过一对反引号（``）来调用（反引号的按键在全尺寸的美式键盘上数字键1的左边。请注意，不要把反引号和单引号“'”搞混了）。我们要运行的外部命令是*perldoc -u -f atan2*。请在命令行上键入这条命令，看看它会输出什么信息。*perldoc*命令在大部分系统上都有，可以用它来阅读Perl及其相关扩展和工具程序的

说明文档[注25]。这条命令会显示在Perl中如何使用三角函数atan2的一些信息。不过这里我们只是用它来示范如何处理外部命令的输出结果。

当反引号里的命令执行完毕后，输出结果会一行行依次被存储在@lines这个数组变量中。后面一行代码会启动一个循环，依次对每行数据进行处理。循环里的代码是缩排过的，虽然Perl并不强迫你这么做，但好的程序员都会如此要求自己。

循环里的第一行代码看起来最恐怖：`s/\w<([^>]+)>/\U$1/g;`。此处不会深入讨论细节，它的大概意思就是：对每个包含一对尖括号（<>）的行进行相应的数据替换操作。而在*perldoc*命令的输出结果里，应该至少有一行符合此操作条件。

至于循环内的第二行代码则来了个大变样，一下简洁不少，它直接输出每行的内容（有可能被上面的替换操作修改过）。最后的输出结果看起来应该和*perldoc -u -f atan2*的执行结果差不多，只是其中出现尖括号的地方有所不同。

就这样，在短短数行代码中，我们调用了别的程序，并将它的输出结果放到内存，然后更新内存里的数据，最后输出。很多时候，我们都是用Perl来做这种数据转换工作的。

习题

一般来说，每章结束时都会有几道习题，解答则放在附录A中。不过在本章的习题中就不用写程序了——答案已经在前面出现过了。

如果在你的机器上不能做这些习题，请先自己详细检查几遍，然后再就近询问专家。别忘了有时候我们需要修改下程序，弱化一部分功能再运行，就像我们在文中提到的那样：

1. [7]键入前面的“Hello, world”程序，想办法让它运行起来！程序文件可以取任何名称，比如说*ex1-1*这样简洁的名称就不错，用以表示第一章中的第一道习题。一般有经验的程序员都会如此命名。这个练习的目的是要确保你的系统上已经准备好开发环境，如果这个程序能正常运行，那就说明系统中的*perl*可以正常工作。

注25： 如果没有*perldoc*命令可用，很可能意味着你的系统没有命令行界面，并且你的Perl也无法通过反引号运行外部命令（如*perldoc*）或打开一个管道传送数据给外部程序，有关管道方面的知识可参考第十四章。要是碰上这种情况，现在就请先跳过*perldoc*这行代码吧。

2. [5]在命令行提示符后键入*perldoc -u -f atan2*这条命令，并观察它的输出结果。如果无法执行，请就近询问系统管理员或参考Perl发行版本的说明，找出如何调用*perldoc*或相近的其他命令（下道题目也会用到这个命令）。

3. [6]键入第二个例子中给出的程序（参阅上一节），看看它会输出什么。提醒一下：请仔细按照书上的标点符号键入，不要打错了！你能看出来它是如何改变外部命令的输出结果的吗？

第二章

标量数据

英语跟许多其他语言一样区别单数（singular）和复数（plural）。作为一个由人类语言学家设计的计算机语言，Perl也有类似的区别。一般来说，Perl用标量（*scalar*）[注1]来称呼单个事物。标量是Perl里面最简单的一种数据类型。对大部分标量来说，它要么是数字（比如255或是3.25e20），要么是由字符组成的序列（比如`hello`[注2]或林肯总统的Gettysburg演讲词）。虽然你可能会认为数字和字符串是两码事，但对Perl来讲，这两者大多情况下都是可以在内部转换的。

你可以用操作符对标量进行操作（比如加法或字符串连接），产生的结果通常也是一个标量。标量可以存储在标量变量里，也可以从文件和设备读取或者写入这些位置。

数字

虽然标量不外乎数字和字符串这两种情况，但我们还是分别讨论，这样比较容易厘清头绪。我们先讨论数字类型的标量，然后再来讨论字符串类型的标量。

所有数字的内部格式都相同

接下来，我们会看到如何设定整数（不带小数点的数字，比如255和2 001）以及浮点数

注1：　这和数学或物理学上同名的“标量”没什么关系，Perl语言里是没有“向量（vector）”的。

注2：　如果用过其他程序语言，你可能会认为`hello`是5个字符的集合，而非单个事物。但在Perl里，字符串就是独立的一个标量值。当然，有必要时我们还是可以访问其中的每个字符的，具体做法会在后面的章节介绍。

（带有小数点的数字，比如3.14159或者1.35×10^{25}等）。但在Perl内部，则总是按“双精度浮点数（double-precision floating-point）[注3]”的要求来保存数字并进行运算的。也就是说，Perl内部并不存在整数值——程序中用到的整型常量会被转换成等效的浮点数值[注4]。当这类转换发生时，你大概无法注意到（也许根本不想注意），不过如果你想找找看有没有专门的整型运算（相对于浮点运算），请马上停手，因为根本没有这种东西[注5]。

浮点数直接量

直接量（literal）是指某个数字在Perl源代码中的写法。直接量并非运算结果，也不是I/O（输入／输出）操作的结果，它只是直接键入程序源代码中的数据。

你应该对Perl浮点数直接量的写法很熟悉了。小数点与前置的正负号都是可选的，数字后面也可以加上用“e”表示的10的次方标识符（即指数表示法）。例如下面这些不同写法：

```
1.25
255.000
255.0
7.25e45      # 7.25乘以10的45次方（一个很大的数）
-6.5e24      # 负6.5乘以10的24次方
             #（一个非常大的负数）
-12e-24      # 负12乘以10的-24次方
             #（一个非常小的负数）
-1.2E-23     # 另一种表示法：字母E也可以是大写的
```

整数直接量

整数直接量同样简单易懂，例如：

```
0
2001
-40
```

注3：　此处所谓的双精度浮点数，就是当初用来编译Perl的C编译器的double类型。虽然它的大小可能因机器而异，但是大部分现代系统都会用IEEE-745的格式，它能表示15位的精度，有效值的范围从1e-100到1e100。

注4：　其实Perl内部有些时候也会使用整数，只是不在程序员的控制范围中。这也就是说，你唯一能察觉的，就是程序运行好像快了那么一点。没有人会抱怨这种好事吧？

注5：　好吧，是有个integer编译命令（pragma）。没错，但它已超出了本书的讨论范围。当然，某些运算会将浮点数强制转换成整数，我们稍后会谈到。不过这跟我们前面讲的不是一回事。

```
255
61298040283768
```

最后一个数字读起来有些费力。Perl允许你在整数直接量中插入下划线，将若干位数分开，写成这样看起来就很清楚了：

```
61_298_040_283_768
```

这两种写法都表示同一个数字，只是写法不同而已。你也许会觉得用逗号分隔看上去更自然一些，但你要知道，逗号在Perl里已经有更重要的用途（我们会在第三章里看到），所以为了避免产生歧义，这里只能用下划线。

非十进制的整数直接量

和许多其他程序语言一样，Perl也允许使用十进制（decimal）以外的其他进制来表示数字。八进制（octal）直接量以0开头，十六进制（hexadecimal）直接量以0x开头，而二进制（binary）直接量则以0b开头[注6]。十六进制数的A到F（或是小写的a到f也行），代表十进制数的10到15的数字。例如：

```
0377          # 八进制的377，等于十进制的255
0xff          # 十六进制的FF，也等于十进制的255
0b11111111    # 也等于十进制的255
```

虽然这三个数字看起来并不相同，但对Perl而言都是同一个数字。既然0xFF和255.000在Perl里没有分别，那么采用哪种写法最能说明问题，我们就该用哪种，这样对你自己或者该程序将来的维护人员都有好处（其实看不懂代码逻辑的可怜虫很多，但这个可怜虫往往就是你自己：在三个月之后，你可能完全记不起当时为什么要这么写）。

非十进制直接量的长度超过4个字符时，读起来可能会有些困难。因此Perl同样容许在这些直接量中使用下划线，分开后读起来更容易些：

```
0x1377_0B77
0x50_65_72_7C
```

注6：“前置零（leading zero）”的表示法只对直接量有效——不能用于字符串到数字的自动转换，关于这一点我们稍后会在本章的“数字与字符串之间的自动转换”一节看到。但你可以用oct()和hex()这两个内置函数，把看起来像八进制或十六进制的直接量转换成数字。虽然没有用来转换二进制直接量的bin()函数，但是oct()可以接受以0b开头的字符串。

数字操作符

Perl提供了各种常见的数字运算操作符，比如加法、减法、乘法、除法等等。请看示例：

```
2+3          # 2加上3，得5
5.1-2.4      # 5.1减去2.4，得2.7
3*12         # 3乘以12，得36
14/2         # 14除以2，得7
10.2/0.3     # 10.2除以0.3，得34
10/3         # 总是按浮点类型进行除法运算，所以得3.3333333...
```

Perl还支持取模（*modulu*）操作符（%）。表达式10%3的结果是1，也就是10除以3的余数。取模操作符先取整然后再求余，所以10.5%3.2和10%3的计算结果是相同的[注7]。另外，Perl也提供类似FORTRAN语言的乘幂（*exponentiation*）操作符，满足了许多Pascal和C用户的心愿。乘幂操作符以双星号表示，比如2**3代表2的3次方，计算结果为8[注8]。此外还有一些别的数字操作符，我们会在之后用到时再做介绍。

字符串

字符串就是一个字符序列，比如hello或者☃★๛。字符串可以各种字符任意组合而成[注9]。最短的字符串不包含任何字符，所以也叫做空字符串。最长的字符串的长度没有限制，它甚至可以填满所有内存（与此同时你也无法再对它进行任何操作）。这符合Perl尽可能遵循的“无内置限制（nobuilt-inlimits）”的原则。字符串通常是由可输出的字母、数字及标点符号组成，其范围介于ASCII编码的32到126之间。不过，因为字符串能够包含任何字符，所以可用它来创建、扫描或操控二进制数据，这是许多其他工具语言望尘莫及的。比如，你可以将一个图形文件或编译过的可执行文件读进Perl的字符串变量，修改它的内容后再写回去。

Perl完全支持Unicode，所以在字符串中可以使用任意一个合法的Unicode字符。不过由于Perl的历史原因，它不会自动将程序源代码当作Unicode编码的文本文件读入，所以如果你想要在源代码中使用Unicode书写直接量的话，得手工加上utf8编译指令[注10]：

注7：　要注意的是，进行取模运算时，如果其中一边或两边都是负数，则不同的Perl版本可能会得出不同的结果。

注8：　一般来说，你不能计算负数的负数次方。数学极客（geek）都知道，这样算出来的结果将会是复数（complex number）。如果要使用复数的话，必须借助Math::Complex模块。

注9：　和C或C++语言不同，空字符（NUL）在Perl里并没有特殊意义。Perl会另行记住字符串的长度，而不是用空字符来表示字符串的结尾。

注10：　最好养成习惯始终加上这句，除非有明确原因指出不该这么做。

```
use utf8;
```

对于本书后续的章节，我们都假设你使用这条编译指令。对有些代码来说实际上没啥影响，不过要是你看到源代码中出现ASCII字符范围以外的字符，则说明必须加上这条编译指令。此外，你还得确保以UTF-8编码的方式保存文件。如果之前跳过了前言中有关Unicode的建议说明的话，现在不妨去读一下附录C中更多与Unicode相关的内容。

和数字一样，字符串也有直接量记法，也就是Perl程序中字符串的书写方式。字符串直接量有两种不同形式：单引号内的字符串和双引号内的字符串。

单引号内的字符串直接量

单引号内的字符串直接量（*single-quoted string literal*）指的是一对单引号（'）圈引的一串字符。前后两个单引号并不属于字符串的内容，它们只是用来让Perl识别字符串的开头与结尾。除了单引号和反斜线字符外，单引号内所有字符都代表它们自己（包括换行符，如果该字符串表示多行的数据的话）。要表示反斜线字符本身，需要在这个反斜线字符前再加一个反斜线字符表示转义；要表示单引号本身时，同样在单引号前加一个反斜线字符表示转义。来看看具体的例子：

```
'fred'        # 总共4个字符：f、r、e和d
'barney'      # 总共6个字符
''            # 空字符串（没有字符）
'⅚∞☃☠        # 某些“宽”Unicode字符
'Don\'t let an apostrophe end this string prematurely!'
'the last character is a backslash: \\'
'hello\n'     # hello后面接着反斜线和字母n
'hello
there'        # hello、换行符、there（总共11个字符）
'\'\\'        # 单引号紧接着反斜线（总共2个字符）
```

请注意，单引号内的\n并不是换行符，而是表示字面上的两个字符：反斜线和字母n。只有在反斜线后面接续单引号或者反斜线时，才表示转义。

双引号内的字符串直接量

双引号内的字符串直接量（*double-quoted string literal*）同样也是字符串序列，只不过这次换成双引号表示首尾。不过双引号中的反斜线更为强大，可以转义许多控制字符，或是用八进制或十六进制写法来表示任何字符。还是来看一些具体例子：

```
"barney"            # 和'barney'写法一样的效果
"hello world\n"     # helloworld，后面接着换行符
"The last character of this string is a quote mark: \""
"coke\tsprite"      # coke、制表符（tab）和sprite
```

```
"\x{2668}"          # Unicode中名为HOT SPRINGS的字符的代码点（code point）
```

请注意，对Perl来说双引号内的字符串直接量"barney"和单引号内的字符串直接量'barney'是相同的，都是代表那6个字符组成的字符串。这和前面提到的数字直接量的情况相类似，0377只不过是255.0的另一种等效写法。哪种写法更为自然流畅，能表示实际意义，就可以选择哪种方式，Perl给你这样的自由。一般来说，要通过反斜线转义表示换行符（\n）那样的特殊字符的话，就该用双引号书写直接量。

反斜线后面跟上不同字符，可以表示各种不同的意义（一般我们把这种借助反斜线组合表示特殊字符的方法称作反斜线转义）。在双引号内的字符串直接量内允许使用的比较完整的转义字符清单如表2-1所示。

表2-1：双引号内字符串的反斜线转义

组合	意义
\n	换行
\r	回车
\t	水平制表符
\f	换页符
\b	退格
\a	系统响铃
\e	Esc（ASCII编码的转义字符）
\007	八进制表示的ASCII值（此例中007表示系统响铃）
\x7f	十六进制表示的ASCII值（此例中7f表示删除键的控制代码）
\x{2744}	十六进制表示的Unicode代码点（这里的U+2744表示雪花形状的图形字符）
\cC	控制符，也就是Control键的代码（此例表示同时按下Ctrl键和C键的返回码）
\\	反斜线
\	双引号
\l	将下个字母转为小写的
\L	将它后面的所有字母都转为小写的，直到\E为止
\u	将下个字母转为大写的
\U	将它后面的所有字母都转为大写的，直到\E为止
\Q	相当于把它到\E之间的非单词（non word）字符加上反斜线转义
\E	结束\L、\U和\Q开始的作用范围

双引号内字符串的另一种特性称为变量内插（*variable interpolated*），这是指在使用字

符串时，将字符串内的变量名称替换成该变量当前的值。因为我们还没有正式介绍什么是变量，所以这部分内容稍后再谈。

字符串操作符

字符串可以用.操作符（没错，就是句点符号）连接起来。两边的字符串都不会因此操作而被修改，就像2+3的运算不会改变2或3一样。运算后得到一个更长的字符串，可以继续用于其他运算或赋予某个变量。来看具体例子：

```
"hello" . "world"           # 等同于"helloworld"
"hello" . ' ' . "world"     # 等同于'hello world'
'hello world' . "\n"        # 等同于"hello world\n"
```

要注意的是，连接运算必须显式使用连接操作符（*concatenation operator*），而不像其他某些语言只需把两个字符串放在一起就行。

还有个比较特殊的字符串重复操作符（*string repetition operator*），它其实就是一个小写字母x。此操作符会将其左边的操作数（也就是要重复的字符串）与它本身重复连接，重复次数则由右边的操作数（某个数字）指定。来看具体例子：

```
"fred" x 3          # 得"fredfredfred"
"barney" x (4+1)    # 得"barney"×5,亦即"barneybarneybarneybarneybarney"
5 x 4.8             # 本质上就是"5"x4，所以得"5555"
```

最后一个例子有必要详加说明。因为重复操作符的左操作数必然是字符串类型，所以数字5在进行重复操作前，先被转换成单字符的字符串"5"（具体转换规则稍后会提到），然后这个新字符串被重复了4次，生成含有4个字符的新字符串5555。注意，如果我们将操作数对调，亦即写成4x5这样，结果会是得到重复5次的字符4，也就是44444。这说明字符串重复操作并不满足交换（commutative）律。

重复次数（右操作数）在使用前会先取整（4.8变成4）。重复次数小于1时，会生成长度为零的空字符串。

数字与字符串之间的自动转换

通常Perl会根据需要，自动在数字和字符串之间进行类型转换。那它究竟是如何知道需要数字还是字符串呢？这完全取决于操作符。如果操作符（比如+）需要的是数字，Perl就会将操作数视为数字；在操作符（比如.）需要字符串时，Perl便会将操作数视为字符串。因此，你不必担心数字和字符串间的差异，只管合理使用操作符，Perl会自动完成剩下的工作。

对数字进行运算的操作符（比如乘法）如果遇到字符串类型的操作数，Perl会自动将字

符串转换成等效的十进制浮点数进行运算。因此"12"*"3"的结果会是36。字符串中非数字的部分（以及前置的空白符号）会被略过，所以"12fred34" * " 3"也会得出36，而不会出现任何警告信息[注11]。在最极端的情形下，完全不含数字的字符串会被转换成零。比如把"fred"当成数字来用就属于这种情况。

“前置零”的技巧只对直接量有效，不能用于字符串的自动转换，自动转换总是按照十进制数字来处理的[注12]。

```
0377     # 十进制数字255的八进制写法
'0377'   # 会转换成十进制数字377
```

同样地，需要字符串的操作符（比如字符串连接符）意外得到数字时，该数字就会被转换为形式相同的字符串。比如要把字符串Z与“5乘以7的结果”相连接[注13]，写起来非常简单：

```
"Z" . 5 * 7  # 等同于"Z".35，得"Z35"
```

总的来说，大多数时候你根本不必关心数字和字符串的区别，Perl会自动完成转换数据的工作[注14]。

Perl的内置警告信息

当发现程序有些不对劲时，可以让Perl发出警告。从Perl 5.6开始，我们可以通过编译指令开启警告功能（但请注意，需要兼顾早期版本用户时就不能用这种方式）[注15]：

```
#!/usr/bin/perl
use warnings;
```

注11：　除非你要求Perl显示警告信息，稍后我们就会提到有关警告信息的处理。

注12：　碰到以八进制，或者十六进制，甚至二进制写法表示的字符串时，得用专门的转换函数oct()或hex()将其转换成对应数字。详细信息请参阅第252页的“非十进制数字字符串的转换”一节。

注13：　我们很快就会提到优先级和括号。

注14：　如果担心效率的话，大可放心，Perl通常会记住第一次转换的结果，所以实际的转换只是进行一次而已。

注15：　实际上，warnings编译指令能够指定代码的作用范围，详细信息可以参阅*perllexwarn*文档。warnings比-w更灵活，它可以选择只对文件中使用了该编译指令的部分代码开启警告功能，而 -w选项则不加区分，对整个程序中涉及的所有代码都开启警告功能。

也可以在命令行上使用-w选项对要运行的程序开启警告功能[注16]：

```
$ perl -w my_program
```

也可以在shebang行上指定命令行选项：

```
#!/usr/bin/perl-w
```

即便是在非Unix系统上也可以这么用，这和加不加Perl的具体路径没关系，一般都会写成：

```
#!perl -w
```

现在，如果你把'12fred34'当数字用，Perl就会发出警告：

```
Argument "12fred34" isn't numeric
```

虽然发出了警告，但Perl仍然会按照它既定的默认规则，把非数字字符串'12fred34'转换为12。

当然，警告信息通常是给程序员看的，而不是最终用户。要是连程序员都不看的话，警告信息大概也没什么用。另外，警告信息并不会改变程序的行为，只会让它偶尔抱怨几声。如果看不懂某个警告信息，可以利用diagnostics这个编译命令报告更为详尽的问题描述。在*perldiag*文档中列有简要警告信息和详细诊断说明，该文档是理解diagnostics输出信息的最佳参考：

```
#!/usr/bin/perl
use diagnostics;
```

在把use diagnostics这个编译命令加进程序之后，你可能会觉得程序启动好像有点慢。这是因为程序正在忙着加载警告和详细说明到内存，准备好碰到有错误或警告发生，就立即输出相关的错误信息。其实反过来看，这也提示了一种优化程序的方法：如果熟悉各种警告信息的意义，就不必在运行时发出详细的警告解释，去掉use diagnostics这个编译命令就会让程序启动变快，内存消耗也随之减少。（当然，如果能修改好程序，让它不再产生烦人的警告信息，就再好不过了。或者干脆直接关闭警告信息。）

更进一步，这种优化也可以通过Perl的命令行选项-M来实现。与其每次修改程序代码，不如仅在需要时再加载diagnostics编译命令：

```
$perl -Mdiagnostics ./my_program
Argument "12fred34" isn't numeric in addition (+) at ./my_program line 17 (#1)
```

注16： 如果程序用到别人写的模块，那么警告同样会作用于这些模块中的代码，所以使用这种方式也许会看到别人的代码的警告。

```
(W numeric) The indicated string was fed as an argument to
an operator that expecteda numeric value instead. If you're
fortunate the message will identify which operator was so unfortunate.
```

注意警告信息中出现的（W numeric），其中W的意思是警告级别属于普通警告，numeric的意思是警告类型属于数字操作一类。所以，看到这两条就知道潜在问题大致出在哪里。

随着后续深入介绍，我们还会看到关于其他类型错误的警告。不过请记住，将来的Perl也许会因为内部工作机制的变化而令发出警告的方式和内容也随之发生变化。

标量变量

所谓变量（*variable*），就是存储一个或多个值的容器的名称。而标量变量，就是单单存储一个值的变量。后续章节我们还会看到其他类型的变量，比如数组和哈希，它们都可以存储多个值。变量的名称在整个程序中保持不变，但它所持有的值是可以在程序运行时不断修改变化的。

你大概猜到了，标量变量存储的是单个标量值。标量变量的名称以美元符号开头，这个符号也称为魔符（*sigil*），然后是变量的Perl标识符：由一个字母或下划线开头，后接多个字母、数字或下划线。标识符是区分大小写的：变量$Fred和$fred是两个完全不同的变量。不同的大小写字母、数字以及下划线构成了不同的标识符，所以下面的变量各不相同：

```
$name
$Name
$NAME

$a_very_long_variable_that_ends_in_1
$a_very_long_variable_that_ends_in_2
$A_very_long_variable_that_ends_in_2
$AVeryLongVariableThatEndsIn2
```

Perl并不限于使用ASCII字符作为变量名。如果启用了utf8编译指令，那么可用于表示字母或数字的字符会多许多，所以拿它们来作为变量名也是可以的：

```
$résumé
$coördinate
```

Perl通过变量标识符前的魔符来区分它是什么类型的变量。所以不管你取什么名字，都不会和Perl自带的函数或操作符的写法相冲突。

此外，Perl是通过该魔符来判断该变量的使用意图。$的确切意思是“取单个东西”或者“取标量”。因为标量变量总是存储一项数据，所以它的意思就总是取得其中的“单

个”值。在第三章中，你会看到“取单个东西”的魔符应用于其他类型（数组）变量的情况。

给变量取个好名字

一般来说，变量名称应该能说明它的用途和意义。比如变量$r，天知道它代表什么，但$line_length就不同了，一看就知道表示行宽。如果变量只是在接下来的两三行里用到，那么类似$n这样的简短名称倒也无妨，但用于整个程序范围的变量就该取个比较有意义的能说明问题的名字[注17]。

同样的道理，适当补充下划线也能改善变量名的可读性，尤其是在代码维护者的母语和你的不同的情况下。举例来说，$super_bowl这个名称就比$superbowl（超级杯）要好，因为后者也可以理解为$superb_owl（雄伟的猫头鹰）。再比如$stopid，它的意思到底是$sto_pid（存储某个进程的进程号？）、$s_to_pid（转换什么东西为进程号？）、$stop_id（某种“停止”对象的代号？），或者仅仅是把笨蛋（stupid）拼错了？

Perl程序里面的大部分变量名称都习惯使用全小写，正如你在本书中看到的例子一样。只有少数几种情况中才会用到大写字母。而使用全大写的（比如$ARGV）变量一般都是表示特殊意义的变量。如果变量名不止一个单词，有人喜欢用下划线分开，如$underscores_are_cool，也有人喜欢用$giveMeInitialCaps这种风格。怎么选择是你的喜好，不要混用就可以了[注18]。当然你也可以使用全部大写的变量名，但这么一来就有可能和Perl保留的特殊变量的名称相冲突。所以最好还是不要用全大写的名称[注19]。

当然，名称的好坏其实对Perl来说并无差别。你可以把程序中最重要的三个变量取名为$OOO000OOO、$OOO00OOOO和$OOO0O0O0O，Perl不会觉得这是种挑战——但拜托不要找我们来帮你维护程序！

标量的赋值

对标量变量最常见的操作就是赋值（*assignment*）了，也就是将某个值存进变量中。Perl的赋值操作符为等号（这和其他程序语言差不多），等号的左边是变量名称，右边为某个表达式，对表达式求值的结果作为赋予变量的值。来看具体的例子：

注17：　你之所以熟悉自己的程序，无非是因为你创造了它。对你来说，$srly的意思可能是显而易见的，但对其他人来讲就不是那么回事了。

注18：　在*perlstyle*文档中有一些关于变量命名的建议，不妨读一下。

注19：　可以到*perlvar*文档中查阅所有Perl特殊变量的名称。

```
$fred   = 17;          # 将$fred的值设为17
$barney = 'hello';     # 将$barney的值设为5个字符组成的字符串'hello'
$barney = $fred+3;     # 将$barney设为$fred当前值加上3后的结果，即20
$barney = $barney*2;   # $barney现在被设成$barney当前值乘以2后的结果，即40
```

请注意，最后一行中`$barney`变量出现了两次：第一次是取值（在等号右方的表达式中），第二次是赋值（在等号左方），表示要将右方表达式的运算结果存到该变量中。这样写不但合法、安全，而且还十分常见。事实上，正因为这种用法太常见了，我们还有种更简便的书写方式，请看下节。

双目赋值操作符

我们经常会用到类似`$fred = $fred + 5`这种形式的表达式（同样的变量出现在赋值操作符的两边），于是Perl（同C或Java语言一样）提供了更新变量内容的简写方式——双目赋值操作符（*binary assignment operator*）。几乎所有用来求值的双目操作符都可以接上等号，成为相应的双目赋值操作符。比如下面这两行其实是等效的：

```
$fred  = $fred + 5;   # 不使用双目赋值操作符
$fred += 5;           # 使用双目赋值操作符
```

以下这两行也是等效的：

```
$barney  = $barney * 3;
$barney *= 3;
```

以上的例子里，双目赋值操作符都是以某种方式直接修改变量的值，而非对表达式求值后覆盖原变量值。

另一个常见的双目赋值操作符是由字符串连接操作符（`.`）改进而成的追加操作符（`.=`）：

```
$str  = $str. " ";    # 在$str末尾追加一个空格字符
$str .= "";           # 用追加操作符做同样的事
```

几乎所有的双目操作符都可以这么用。比如，乘幂操作符可以改成`**=`，所以`$fred**=3`的意思就是说，“将`$fred`里的值自乘3次（也就是取三次方），再存回`$fred`”。

用print输出结果

一般我们都想要程序输出些什么信息来，否则，也许会有人以为它什么事都没做。`print`操作符就是用来完成这项任务的：它可以接受标量值作为参数，然后不经修饰地将它传送到标准输出（standard output）。除非特别指定，否则一般默认的“标准输出”指的就是终端屏幕。例如：

```
print "hello world\n"; # 输出hello world，后面接着换行符

print "The answer is";
print 6 * 7;
print ".\n";
```

你也可以用print输出一系列用逗号隔开的值：

```
print "The answer is ", 6 * 7, ".\n";
```

这实际上就是一个列表（*list*），但我们现在还没提到列表，稍后再作说明。

字符串中的标量变量内插

一般我们用双引号圈引字符串的目的，除了是要用之前提到过的反斜线转义外，多半是为了使用变量内插（*variable interpolation*）[注20]。其实说白了，就是把字符串内出现的所有标量变量[注21]替换成该变量当前的值。比如：

```
$meal   = "brontosaurus steak";
$barney = "fred ate a $meal";        # $barney现在是"fred ate a brontosaurus steak"
$barney = 'fred ate a '.$meal;       # 另一种等效写法
```

正如上面最后一行所示，不用双引号也可以达成相同效果，但用双引号写起来更简便清晰。

如果标量变量从未被赋值过[注22]，就会用空字符串来替换：

```
$barney = "fred ate a $meat"; # $barney就会变成"fred ate a"
```

如果只是要打印这个变量值，则不必使用变量内插的方式：

```
print "$fred";        # 双引号是多余的
print $fred;          # 这样写比较好
```

在单个变量两边加上引号也不算错[注23]，不过这么写的话，别的程序员可能会在背后嘲笑你哦，或者当面嘲笑也说不定。变量内插又被称为双引号内插，因为它通常是在双引号（而非单引号）里起作用的。在Perl里还有其他一些字符串内插的情况，具体细节我们稍后遇到时会谈到。

注20： 这和数学或统计学上的“插值（interpolation）”无关。

注21： 除了标量变量外，还有其他类型的变量也可以内插，我们稍后会再说明。

注22： 实际上该变量拥有一个特殊的值undef，表示未定义，本章稍后会详细说明。如果启用警告，Perl会抱怨在内插时使用未定义的变量。

注23： 至少，这会把变量值作为字符串处理，而不是数字。只有极少数情况下需要借助这种写法，很多时候这只是在浪费打字时间。

如果要将美元符号本身放进双引号内的字符串，可以在它前面用反斜线转义，以避开它的特殊意义：

```
$fred = 'hello';
print "The name is \$fred.\n";        # 会输出$符号
```

当然，也可以跳开这个变量，用串接的方式构造新的字串：

```
print 'The name is $fred' . "\n";     # 效果相同
```

进行内插时，Perl会尽可能使用最长且合法的变量名称。要是你想在内插的值后面紧接着输出字母、数字或下划线，可能会碰上麻烦[注24]。

当Perl检查变量名称时，它会违背你的本意，将后面的字符当作变量名称的一部分。解决办法很简单，和shell脚本一样，Perl里面我们可以用一对花括号将变量名围起来以避免歧义。要不然，可以先把字符串拆成两半，再利用连接操作符拼接起来：

```
$what = "brontosaurus steak";
$n = 3;
print "fred ate $n $whats.\n";         # 不是steaks，而是$whats的值
print "fred ate $n ${what}s.\n";       # 现在用的是$what的值
print "fred ate $n $what" . "s.\n";    # 另一种写法，但比较麻烦
print 'fred ate ' . $n .' ' . $what."s.\n";#特别麻烦的写法
```

借助代码点创建字符

有时候我们需要输入键盘上没有的那些字符，比如é、å、α或者א等。取得这些字符的方法得看用的是什么系统的输入法或者哪一款文本编辑器。不过，与其费力寻找字型输入，还不如直接键入这些字符的代码点（code point）[注25]，再通过chr()函数转换成对应字符来得方便：

```
$alef  = chr( 0x05D0 );
$alpha = chr( hex('03B1') );
$omega = chr( 0x03C9 );
```

反过来，我们可以通过ord()函数把字符转换为代码点：

注24：　另外还有一些可能会出问题的字符。在标量变量名称后面如果需要接上左方括号或左方括号，请在括号前面加上反斜线。要是变量名称后面接的是单引号或两个冒号，也需要以同样的方式处理。要不然，你也可以使用正文里提到的花括号表示法来代替。

注25：　Unicode的概念将贯穿本书，所以我们一直会提到有关代码点（code point）的说法。如果只是在ASCII中，则只需要通过表示序数（*ordinal value*）的数字说明字符。如果你对Unicode背景知识还不熟悉的话，请即刻参阅附录C。

```
$code_point=ord('? ');
```

通过代码点创建的字符同样可用于双引号内的变量内插：

```
"$alpha$omega"
```

如果不预先创建变量，也可以直接在双引号内用\x{}的形式表示，虽然看起来有点眼花，但也还算方便：

```
"\x{03B1}\x{03C9}"
```

操作符的优先级与结合性

在复杂的表达式里，先执行哪个操作再执行哪个操作，取决于操作符的优先级。比如在表达式2+3*4中，先算加法还是乘法？如果先算加法，会得到5*4，也就是20；如果先算乘法（就像数学课里教的），会得到2+12，也就是14。还好，Perl的选择和一般的数学计算相同，是先算乘法。也就是说，乘法的优先级高于加法。

你可以用括号（即圆括号）来改变执行优先级。任何放在括号里的运算都比括号外的优先级高（和数学课里学到的一样）。因此，如果真的想让加法比乘法先计算，可以写成(2+3)*4，得20；当然也可以加上多余的括号，像2+(3*4)，以此强调乘法比加法先计算的事实。

在加法和乘法的例子里，优先级相当清楚，不言自明。但当我们碰到像字符串连接和乘幂计算时，就很难断定谁先谁后了。正确的解决之道不外乎参考*perlop*文档中标准的Perl操作符优先级表，我们摘取其中主要部分列于表2-2中[注26]。

表2-2：操作符的结合性与优先级（从高至低排序）

结合性	操作符
左	括号；给定参数的列表操作符
左	->
	++ --（自增；自减）
右	**
右	\ ! ~ + -（单目操作符）
左	=~ !~

注26： 给C程序员一个好消息：所有同时在Perl和C里出现的操作符，它们的优先级和结合性都是相同的。

表2-2：操作符的结合性与优先级（从高至低排序）（续）

结合性	操作符
左	* / % x
左	+ - .（双目操作符）
左	<< >>
	具名的单目操作符（-X文件测试；rand）
	< <= > >= lt le gt ge（“不相等”操作符）
	== != <=> eq ne cmp（“相等”操作符）
左	&
左	| ^
左	&&
左	||
	.. ...
右	?:（三目操作符）
右	= += -= .=（以及类似的赋值操作符）
左	, =>
	列表操作符（向右结合）
右	not
左	and
左	or xor

在这个表格里，任何操作符的优先级都高于列在它下方的所有操作符，并低于列在它上方的所有操作符。如果操作符间的优先级相同，则按照结合性的规则来判断。

当两个优先级相同的操作符抢着使用三个操作数时，优先级便交由结合性解决：

```
4 ** 3 ** 2 # 4 ** (3 ** 2)，得4 ** 9（向右结合）
72 / 12 / 3 # (72 / 12) / 3，得6/3，得2（向左结合）
36 /  6* 3  # (36/6)*3，得18
```

在第一个例子里，因为**操作符是向右结合的，所以隐含的括号便放在右边；而*和/是向左结合的，因此隐含的括号便放在左边。

那么，是不是该把优先级表背下来呢？不！没人会这么做。要是记不起顺序又懒得查表，直接用括号明确就是了。毕竟，要是你在没有括号的情况下会忘记顺序，那么程序维护员也会遇到相同的麻烦。所以，还是对他好一点吧：说不定将来哪天，那个人就是你自己。

比较操作符

对数值进行比较时，Perl的比较操作符类似于代数系统：<、<=、==、>=、>、!=，这个是符合我们常规逻辑的。这些操作符的返回值要么是真（*true*），要么是假（*false*），我们将会在下一节详细讨论这些返回值的含义。这些操作符在Perl中的写法和在其他语言中的写法可能略有不同。例如，“相等”用的是==而不是单个的=，因为=在Perl里表示变量赋值。另外，“不相等”用的是!=，因为<>在Perl里另有其义。而“大于或等于”用的是>=不是=>，因为后者在Perl里也有别的意义。事实上，几乎每一种标点符号的组合在Perl里都有特定的用处。所以呢，如果哪天你的灵感突然告竭，就让猫猫在键盘上走个几圈，再进行调试吧。

想要比较字符串时，Perl有一系列的字符串比较操作符，看起来像是些奇怪的短语：lt、le、eq、ge、gt以及ne。它们会逐一比对两个字符串里的字符，判定它们是否彼此相等或是哪一个排在前面。请注意，字符在ASCII编码或者Unicode编码中的顺序并不总是对应于字符本身意义上的顺序。至于如何修正，可以参考第十四章中的相关内容。

完整的比较操作符（包括用于数字及字符串）列在表2-3。

表2-3：数值与字符串的比较操作符

比较	数字	字符串
相等	==	eq
不等	!=	ne
小于	<	lt
大于	>	gt
小于或等于	<=	le
大于或等于	>=	ge

来看几个用到这些比较操作符的表达式：

```
35 ! = 30 + 5          # 假
35 == 35.0             # 真
'35' eq '35.0'         # 假（当成字符串来比较）
'fred' lt 'barney'     # 假
'fred' lt 'free'       # 真
'fred' eq "fred"       # 真
'fred' eq 'Fred'       # 假
' ' gt ''              # 真
```

if控制结构

学会如何比较两个值后，你可能需要根据比较结果决定下一步流程。Perl和所有同类型的程序语言一样，也具备if条件语句控制结构：

```
if($name gt 'fred') {
   print"'$name' comes after 'fred' in sorted order.\n";
}
```

如果要在条件不符时做别的处理，可以使用else关键字：

```
if($namegt'fred'){
   print "'$name' comes after 'fred' in sorted order.\n";
}else{
   print "'$name' does not come after 'fred'.\n";
   print "Maybe it's the same string, in fact.\n";
}
```

条件语句中的代码块周围一定要加上表示界限的花括号，这点和C语言不同（不管你有没有学过C）。你最好和上面一样，将代码块里的内容向里缩排，这样程序读起来会方便许多。如果你用的是专为程序员设计的编辑器（我们之前在第一章中提到过），这类事情可以由它完成。

布尔值

其实，任何标量值都可以成为if控制结构里的判断条件。如果把表达式返回的真假值保存到变量中，那么在判断时可以直接检查该变量的值，读起来也更方便：

```
$is_bigger = $name gt 'fred';
if ($is_bigger) {...}
```

但Perl是如何决断给定值的真假呢？和其他语言不同，Perl并没有专用的“布尔（Boolean）”数据类型，它是靠一些简单的规则来判断的[注27]：

- 如果是数字，0为假；所有其他数字都为真。
- 如果是字符串，空字符串（''）为假；所有其他字符串都为真。
- 如果既不是数字也不是字符串，那就先转换成数字或字符串再行判断[注28]。

注27：　这并不是Perl内部的完整规则，但足以让你用来进行判断。

注28：　也就是说，undef（我们马上就会看到）表示假，而且所有的引用（关于引用，我们在另一本书《Intermediate Perl》中有讨论）都是真。

其实上面的规则中还隐含着一个技巧。字符串'0'跟数字0是同一个标量值，所以Perl会将他们一视同仁。也就是说，字符串'0'是唯一被当成假的非空字符串。

要取得任何布尔值的相反值，可以用!这个单目取反操作符。若它后面的操作数为真，就返回假；若后面的操作数为假，则返回真：

```
if (! $is_bigger) {
    # 如果$is_bigger不为真，则执行这里的代码
}
```

这里还有一个小技巧，由于!会颠倒真假值，并且Perl又没有专门的布尔类型的变量，所以!总是会返回某个代表真假的标量值。而数字1和undef都是非常自然的表示真假的标量值，所以人们常常喜欢把布尔值归一化到这两个值来表示。转换过程只是利用了连续两次的!反转操作，得到表示布尔值的变量：

```
$still_true  = !! 'Fred';
$still_false = !! '0';
```

不过，相关文档中从未说明一定就是返回1或undef，但我们觉得将来也应该不会有变化。

获取用户输入

此刻，你大概已经相当好奇，该如何让Perl程序读取从键盘输入的值？最简单的方式就是使用“行输入”操作符<STDIN>[注29]。

只要把<STDIN>放在程序中希望返回标量值的位置上，Perl就会从标准输入（standard input）读取一行文本（直到换行符为止）。标准输入可以有多种意义，但除非另外设定为其他输入来源，否则默认就是调用程序的用户（多半就是你）手边的键盘。如果<STDIN>里没有可供读取的字符（一般都是如此，输入缓存区没有任何数据，除非你在程序启动期间预先打了一整行字符），Perl程序就会停下来，等待你输入某些字符，直到看见换行符（即按下回车键）为止[注30]。

注29：这其实是作用在STDIN文件句柄上的“行输入”操作符，但在我们讨论文件句柄之前（第五章）无法多做解释。

注30：准确地说，其实是你的操作系统在等待输入，而*perl*在等待操作系统的反馈。虽然因系统类型及配置不同而略有差异，但通常都可以在按下回车键之前用退格键来修改错字，因为这时候还是操作系统在处理，尚未转呈给*perl*解释器。如果需要额外的输入控制功能，可以到CPAN上下载Term::ReadLine模块。

由<STDIN>返回的字符串一般在末尾都会带有换行符[注31]。所以，通过下面这段代码，我们可以看到实际发生的情况：

```
$line = <STDIN>;
if ($line eq "\n") {
    print "That was just a blank line!\n";
} else {
    print "That line of input was: $line";
}
```

不过实际编写代码时，很少需要保留末尾换行符，所以人们常常会用chomp()操作符去掉它。

chomp操作符

乍看之下，chomp()操作符的用途好像太过简单专一：只能作用于单个变量，且该变量的内容必须为字符串，如果该字符串末尾是换行符，chomp()的任务就是去掉它。差不多这就是它所有的工作了。比如：

```
$text = "a line of text\n"; # 或者从<STDIN>读进来
chomp($text);               # 去除行末的换行符
```

其实它还是非常有用的，你以后写的每个程序几乎都少不了它。如上所示，处理字符串变量时，它是去除行末换行符的最佳方式。事实上，chomp()还有一种取巧的用法，因为Perl有一条规则：任何需要变量的地方，都可以用赋值运算表达式代替。实际上Perl会先做赋值运算，然后返回赋值后的变量，所以chomp()最常见的用法就是连用：

```
chomp($text = <STDIN>); # 读入文字，略过最后的换行符

$text = <STDIN>;        # 做同样的事……
chomp($text);           # ……却分成两步
```

乍看之下好像采用合并方式的chomp()写起来并不轻松，反而显得更复杂！如果把它看作两次操作（读取一行文字，再对它做chomp()），那么分两步写确实比较自然。但如果将它看作一次操作（只读取文字，不含换行符），那么合并的写法就更自然了。许多Perl程序员都倾向于使用这种写法，所以现在你也该开始习惯起来。

其实，chomp()本质上是函数。而作为一个函数，它就有自己的返回值。chomp()函数的返回值是实际移除的字符数。这个数字几乎没有用处：

注31：　这里有个例外，那就是标准输入流在读入行期间突然中止了。可是，不以换行符结尾的文件当然不能算是正常的文本文件！

```
$food = <STDIN>;
$betty = chomp $food; # 会得到返回值 1——不过我们早就知道了！
```

正如你所见，使用chomp()时，可以加上括号，也可以不加。这是Perl的另一项惯例：除非去掉括号会改变表达式的意义；否则括号可以省略。

如果字符串后面有两个以上的换行符[注32]，chomp()仅仅删除一个；如果结尾处没有换行符，它什么也不做，直接返回零。

while控制结构

Perl和大部分用来实现算法的语言一样，也有好几种循环结构[注33]。在while循环中，只要条件持续为真，就会不断执行代码块：

```
$count = 0;
while ($count < 10) {
    $count += 2;
    print "count is now $count\n"; # 依次打印值 2 4 6 8 10
}
```

这里的真假值与之前提到的if条件测试里的真假值定义相同。代码块外围的花括号也和if控制结构的一样必不可少。条件表达式在第一次执行代码块之前就会被求值，所以如果它一开始就为假，里面的循环就会被直接略过。

undef值

如果还没赋值就用到了某个标量变量，会有什么结果呢？答案是，不会发生什么大不了的事，也绝对不会让程序中止运行。在首次赋值前，变量的初始值就是特殊的undef（未定义）值，它在Perl里的意思仅仅是：这里空无一物——走开、走开。如果你想把这个“空无一物”当成数字使用，它就会表现得像零；如果当成字符串使用，它就会表现得像空字符串。但undef既不是数字也不是字符串，它完全是另一种类型的标量值。

既然undef作为数字时会被视为零，我们可以很容易地构造一个数字累加器，它在开始

注32：　如果是逐行读取输入的，就永远不会发生这种情况。不过，假如我们把输入分隔符（$/）设成换行符以外的值，或者使用read函数读入一连串指定长度的字符串，又或者操作手动拼接起来的字符串时，就有可能碰到连续两个换行符的情况。

注33：　任何程序员都会不小心写出无限循环的程序。如果你不幸碰上这种情况，一般结束这个程序的进程就可以了，就像结束系统上的其他普通进程一样。如果它还在前端运行，一般按下Control+C就可以停止程序运行。至于其他方法，请查阅操作系统的相关说明。

时是空的：

```
# 累加一些奇数
$n = 1;
while ($n < 10) {
    $sum += $n;
    $n += 2; # 准备下一个奇数
}
print "The total was $sum.\n";
```

在循环开始之前，$sum的初始值是undef，但这并不妨碍程序的运行。当第一次执行循环时，$n的值是1，所以循环里的第一行会将$sum的值加上1。这么做，如同将现值为0的变量加1（因为我们把undef当成数字用），所以累加的值会变成1。此后它已经被初始化了，所以就能按常规方式累加。

同样的道理，也可以做出一个字符串累加器，它在一开始时是空的：

```
$string .= "more text\n";
```

如果$string的初始值是undef，它在这里就会被当成空字符串处理，变量的值被设为"moretext\n"。如果它里面已经有字符串，就会把新的文字追加在后面。

Perl程序员常常根据需要把新变量当作零或空字符串来用。

许多操作符在参数越界或不合理时会返回undef。除非有特别处理，否则就会返回零或空字符串。实际上这并不代表有什么大问题，而且有很多程序员就是靠这种特性来写程序的。但你该知道的是，在警告信息开启时，Perl通常会对未定义值的危险用法发出警告，因为那可能就是程序里的缺陷。举例来说，复制某个undef的变量到另一个变量没有什么问题，但若要用print将它输出就会引发警告信息。

defined函数

行输入操作符<STDIN>有时候会返回undef。在一般状况下，它会返回一行文本。但若没有更多输入，比如读到文件结尾（end-of-file）时，它就会返回undef来表示这个状况[注34]。要判断某个字符串是undef而不是空字符串，可以使用defined函数。如果是undef，该函数返回假，否则返回真：

```
$madonna = <STDIN>;
if ( defined($madonna) ) {
    print "The input was $madonna";
```

注34： 一般来说，键盘输入没有“文件结尾”可言；不过，此输入也可能是由文件重定向而来的。另外，用户也有可能按下系统转义的“文件结尾”按键序列。

```
} else {
    print "No input available!\n";
}
```

如果想自己创建undef值，可以直接使用同名的undef操作符：

```
$madonna = undef; # 回到虚无，仿佛从未用过
```

习题

下列习题答案参见第308页上的“第二章习题解答”一节：

1. [5]写一个程序，计算在半径为12.5时，圆的周长应该是多少。圆周长是半径的长度乘上2π（大约是2乘以3.141592654）。计算结果大约是78.5。
2. [4]修改上题的程序，让它提示用户键入半径的长度。当用户键入12.5时，出来的计算结果应该和上题相同。
3. [4]修改上题的程序，当用户键入小于0的半径时，输出0，而不是负数。
4. [8]写一个程序，提示用户键入两个数字（分两行键入），然后输出两者的乘积。
5. [8]写一个程序，提示用户键入一个字符串及一个数字（分两行键入），然后以数字为重复次数，连续输出字符串（提示：使用x操作符）。在用户键入“fred”和“3”时，应该会输出3行“fred”；如果用户键入的是“fred”与“299792”，输出结果应该是一大堆。

第三章

列表与数组

如果说Perl的标量代表的是单数（singular），那正如第二章开头所讲的，在Perl里代表复数（plural）的就是列表和数组。

列表（*list*）指的是标量的有序集合，而数组（*array*）则是存储列表的变量。在Perl里，这两个术语常常混用。不过更精确地说，列表指的是数据，而数组指的是变量。列表的值不一定要放在数组里，但每个数组变量都一定包含一个列表（即便是不含任何元素的空列表）。图3-1所示的就是一个列表，无论它是否存储在某个数组中。

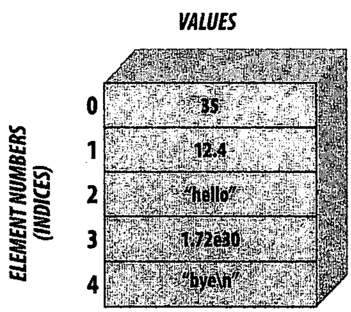


图3-1：包含5个元素的列表

用于列表和数组的操作有许多都是相通的，正如之前看到的标量值和变量一样，所以我们会同时展开对列表和数组的介绍。但不要忘了，本质上两者还是有所区别的。

数组或列表中的每个元素（*element*）都是单独的标量变量，拥有独立的标量值。这些值是有序的，也就是说，从起始元素到终止元素的先后次序是固定的。数组或列表中的每

个元素都有相应的整数作为索引，此数字从0开始递增[注1]，每次加1。所以数组或列表的头一个元素总是第0个元素。

因为每个元素都是独立不相关的标量值，所以列表或数组可能包含数字、字符串、undef值或不同类型标量值的混合。不过最常见的，还是具有相同类型的一组元素，如由书籍标题组成的列表（全都是字符串），或是由余弦函数值组成的列表（全都是数字）。

数组和列表可以包含任意多个元素。最少的情况是没有任何元素，最多的情况则是把可用的内存全部塞满。这种策略又一次体现了Perl“去除不必要的限制”的哲学理念。

访问数组中的元素

如果你曾在其他语言中使用过数组，那么当你看到Perl使用下标数字（subscript）来引用数组元素时，应该会觉得习以为常。

数组元素是以连续的整数[注2]来编号，从0开始，之后的每一个元素依次加1，如下所示：

```
$fred[0] = "yabba";
$fred[1] = "dabba";
$fred[2] = "doo";
```

数组（此例中为"fred"）的名字空间（namespace）和标量的名字空间是完全分开的。你可以在同一个程序里再取一个名为$fred的标量变量，Perl会将两者当成不同的东西，而不会搞混[注3]。（可程序维护员也许会搞混，所以请不要随便将你的变量取相同的名称！）

凡是能够用$fred[2]这类标量变量的地方[注4]，也都可以使用像$fred这样的数组元素。举例来说，你可以取出数组元素的值，或是用第二章中介绍的各种表达式改变它的值：

注1： 跟其他程序语言不同，数组和列表的索引在Perl里总是由0开始。早期的Perl版本允许你改变数组和列表索引值的起始编号（不是针对个别的数组或列表，而是一次性全部生效！）。后来Larry了解到这是个错误的功能，所以目前非常不鼓励（滥）用它。不过，要是你实在好奇，倒是可以看看*perlvar*说明文档中对$[变量的说明。

注2： 是的，还可以用负数表示，我们马上就会看到这种用法。

注3： 这种做法确实在语法上没有二义性，只是有炫耀技术的嫌疑。

注4： 几乎是如此。最明显的例外是foreach循环（我们会在第63页的“foreach控制结构”一节介绍）的控制变量，它必须是简单的标量。还有其他例外，像print、printf使用的“间接对象槽（indirect object slot）”及“间接文件句柄槽（indirect filehandle slot）”。

```
print $fred[0];
$fred[2]  = "diddley";
$fred[1] .= "whatsis";
```

当然，任何求值能得到数字的表达式都可以用作下标。假如它不是整数，则会自动舍去小数，无论正负：

```
$number = 2.71828;
print $fred[$number - 1]; # 结果和print $fred[1]相同
```

假如下标超出数组的尾端，则对应的值将会是undef。这点和一般的标量相同，如果从来没有对标量变量进行过赋值，它的值就是undef：

```
$blank = $fred[ 142_857 ]; # 未使用的数组元素，会得到undef的结果
$blanc = $mel;             # 未使用的标量$mel，也会得到undef的结果
```

特殊的数组索引

假如你对索引值超过数组尾端的元素进行赋值，数组将会根据需要自动扩大——只要有可用的内存分配给Perl，数组的长度是没有上限的[注5]。如果在扩展过程中需要创建增补元素，那么它们的默认取值为undef：

```
$rocks[0]  = 'bedrock';      # 一个元素……
$rocks[1]  = 'slate';        # 又一个……
$rocks[2]  = 'lava';         # 再来一个……
$rocks[3]  = 'crushed rock'; # 再来一个……
$rocks[99] = 'schist';       # 现在有95个undef元素
```

有时候，你会想要找出数组里最后一个元素的索引值。对正在使用的数组rocks而言，最后一个元素的索引值是$#rocks[注6]。但这个数字比数组元素的个数少1，因为还有一个编号为0的元素：

```
$end = $#rocks;                  # 99，也就是最后一个元素的索引值
$number_of_rocks = $end + 1;     # 正确，但后面会看到更好的做法
$rocks[ $#rocks ] = 'hard rock'; # 最后一块石头
```

最后一个例子里，把$#name当成索引值的做法十分常见，所以Larry为我们提供了简写：从数组尾端往回计数的“负数数组索引值”。不过，超出数组大小的负数索引值是不会

注5：　其实严格地说，并不完全如此。最大的数组索引应该是有符号整型数的最大取值，所以实际上你最多只有2 147 483 647个条目。不过从历史经验来看，这么多完全够用。

注6：　这种丑陋的语法来自C shell。还好，我们在实际应用中很少会看到这种写法。

绕回来的。假如你在数组中有3个元素，则有效的负数索引值为-1（最后一个元素）、-2（中间的元素）以及-3（第一个元素）。如果你用-4或者再往后的索引值，只会得到undef而不会绕回到数组尾部。实践中，似乎没有人会使用-1以外的负数索引值：

```
$rocks[ -1 ]   = 'hard rock';   # 和上面最后一个例子相同，但更简单
$dead_rock     = $rocks[-100];  # 得到'bedrock'
$rocks[ -200 ] = 'crystal';     # 严重错误！
```

列表直接量

列表直接量（*list literal*,也就是在程序代码中表示一列数据的写法），可以由圆括号内用逗号隔开的一串数据表示，而这些数据就称为列表元素。例如：

```
(1, 2, 3)      # 包含1、2、3这三个数字的列表
(1, 2, 3,)     # 相同的三个数字（末尾的逗号会被忽略）
("fred", 4.5)  # 两个元素，"fred"和4.5
( )            # 空列表——0个元素
(1..100)       # 100个整数构成的列表
```

上例最后一行用到了..范围操作符（*range operator*），这是我们第一次看到。该操作符会从左边的数字计数到右边，每次加1，以产生一连串的数字[注7]。举例来说：

```
(1..5)              # 与(1, 2, 3, 4, 5)相同
(1.7..5.7)          # 同上，但这两个数字都会被去掉小数部分
(5..1)              # 空列表—— .. 只能向上计数
(0, 2..6, 10, 12)   # 同 (0, 2, 3, 4, 5, 6, 10, 12)
($m..$n)            # 范围由$m和$n当前的值来决定
(0..$#rocks)        # 上节的rocks数组里的所有索引数字
```

正如最后两行所示，数组中的元素不必都是常数——它们可以是表达式，在每次用到这些直接量时都会被重新计算。例如：

```
($m, 17)       # 两个值：$m的当前值以及17
($m+$o, $p+$q) # 两个值
```

当然，列表可以包含任何标量值，像下面这个典型的字符串列表：

```
("fred", "barney", "betty", "wilma", "dino")
```

qw简写

在Perl程序里，经常需要建立简单的单词列表（如同前面的例子）。这时只需使用qw简写，就可以省去键入许多无谓引号的麻烦：

注7：　注意，范围操作符只能从小到大累加。至于生成递减列表，Perl另有办法。

```
qw( fred barney betty wilma dino ) # 同上，但更简洁，也更少击键
```

qw表示“quoted word（加上引号的单词）”或“quoted by whitespace（用空白圈引）”，讲法因人而异。不管怎么说，Perl都会将其当成单引号内的字符串来处理（所以，在qw构建的列表中，不能像双引号内的字符串一样使用\n或$fred）。其中的空白符（如空格、制表符以及换行符）会被抛弃，然后剩下的就是列表的元素。因为空白符会被抛弃，所以上面的列表也可以写成这样（虽不常见）：

```
qw(fred
  barney     betty
wilma dino)  # 同上，但空白符的用法比较奇怪
```

因为qw算是一种引用的形式，所以不能将注释放在qw列表中。有些人喜欢令每个元素单独成行，这样就能排成一列，查看和增删都非常方便：

```
qw(
    fred
    barney
    betty
    wilma
    dino
)
```

前面两个例子是以一对圆括号作为定界符（delimiter），其实Perl还允许你用任何标点符号作为定界符。常用的写法有：

```
qw! fred barney betty wilma dino !
qw/ fred barney betty wilma dino /
qw# fred barney betty wilma dino #   # 看起来像是注释!
```

前后两个定界符也可能不同，如果起始定界符是某种“左”字符，那么结尾定界符必须是对应的“右”字符：

```
qw( fred barney betty wilma dino )
qw{ fred barney betty wilma dino }
qw[ fred barney betty wilma dino ]
qw< fred barney betty wilma dino >
```

如果你需要在被圈引的字符串内使用定界符，那就说明你选错了定界符。不过，在你无法或不希望更换定界符的情况下，还是可以通过反斜线转义来引入这个字符的：

```
qw! yahoo\! google ask msn ! # 将 yahoo! 作为一个元素包含进来
```

和单引号内的字符串一样，两个连续的反斜线表示一个实际的反斜线：

```
qw( This as a \\ real backslash );
```

虽然Perl的座右铭是“办法不止一种（There's More Than One Way To Do It）”，但你可能会纳闷，有谁会需要那么多不同的定界符呢？我们之后会看到，Perl另外还有许多类似的圈引写法，用起来都非常称手。不过就现在来看，如果需要构造一连串的Unix文件名的列表，换作其他定界符就方便多了：

```
qw{
    /usr/dict/words
    /home/rootbeer/.ispell_english
}
```

如果只能以/作为定界符，那么文件路径就会充斥着转义斜线，这个列表会变得相当臃肿难读，并且将来的维护和修改都会很麻烦。

列表的赋值

就像标量值可被赋值给变量一样，列表值也可被赋值给变量：

```
($fred, $barney, $dino) = ("flintstone", "rubble", undef);
```

左侧列表中的三个变量会依次被赋予右侧列表中对应的值，相当于我们分别做了三次独立的赋值操作。因为列表是在赋值运算开始之前建立的，所以在Perl里互换两个变量的值相当容易[注8]：

```
($fred, $barney) = ($barney, $fred); # 交换这两个变量的值
($betty[0], $betty[1]) = ($betty[1], $betty[0]);
```

可是如果（在等号左边的）变量的个数不等于给定的列表值（来自等号右边）的个数时，会发生什么情况呢？对列表进行赋值时，多出来的值会被悄悄忽略掉——Perl认为：如果你真的想要将这些值存放起来的话，你必然会先告知存储位置。另一种情况，如果变量的个数多过给定的列表值的个数，那么那些多出来的变量将会被设成undef[注9]：

```
($fred, $barney) = qw< flintstone rubble slate granite >; # 忽略掉末尾两个元素
($wilma, $dino)  = qw[flintstone];                        # $dino的值为undef
```

明白了列表赋值，你便可以用如下代码来构建一个字符串数组[注10]：

注8：这和C之类的程序语言相反，它们通常没有简单的办法来做这种事。C语言的程序员往往要靠某些宏来达成相同的结果，或者干脆借助临时变量来存储交换值。

注9：嗯，对标量变量来说是这样。数组变量则会变成空列表，稍后我们就会看到。

注10：这里假设rocks数组原本就是空数组。如果之前就有定义并且$rocks[7]有取值，那么此次的赋值运算是不会影响到该元素的。

```
($rocks[0], $rocks[1], $rocks[2], $rocks[3]) = qw/talc mica feldspar quartz/;
```

不过，当你希望引用整个数组时，Perl提供了一个比较简单的记法。只要在数组名之前加上@（at）字符（后面没有检索用的方括号）就可以了。你可以将它读作“all of the（全部的，所有的）”，所以@rocks可以读作“所有的rocks”[注11]。这种写法在赋值操作符的两边都可以使用：

```
@rocks  = qw/ bedrock slate lava /;
@tiny   = ( );                      # 空列表
@giant  = 1..1e5;                   # 包含100 000个元素的列表
@stuff  = (@giant, undef, @giant);  # 包含200 001个元素的列表
$dino   = "granite";
@quarry = (@rocks, "crushed rock", @tiny, $dino);
```

最后一行进行的赋值运算会将@quarry设成拥有5个元素的列表（bedrock、slate、lava、crushed rock、granite），因为@tiny贡献了0个元素给这个列表（请注意，由于空列表里没有任何元素，也就不会有undef被赋值到列表中——但是（如果需要undef）我们也可以显式写明，就像之前对@stuff的操作那样）。此外，值得注意的是，数组名会被展开成（它所拥有的）元素列表。因为数组只能包含标量，不能包含其他数组，所以数组无法成为列表里的元素[注12]。被赋值之前，数组变量的值是空列表，即()。就像标量变量的初始值是undef一样，新的或空的数组的初始值是空列表。

要留意的是，将某个数组复制到另一个数组时，仍然算是列表的赋值运算，只不过这些列表是存储在数组里而已。例如：

```
@copy = @quarry; # 将一个数组中的列表复制到另一个数组
```

pop和push操作符

要新增元素到数组尾端时，只要将它存放到更高的索引值对应的新位置就行了。不过，

注11：Larry宣称他之所以选择$与@这两个符号，是因为$看起来像$calar，即scalar（标量），而@则像@rray，即array（数组）。如果你看不懂，或是不想用这种方式帮助记忆，那也无所谓。

注12：不过在《Intermediate Perl》里你会学到一种称为“引用（reference）”的特殊标量。它能让我们做出“列表的列表”以及其他更有趣或有用的结构。但在那种情况下，也不是真的将列表存进另一个列表中，而是将引用存放到数组里。

真正的Perl程序员是不用索引的[注13]。在随后几个小节中，我们将会介绍一些不用索引对数组进行操作的方法。

我们常把数组当成堆栈（*stack*）来用，比如在数组列表右侧添加新值或者删掉旧值，这有点像咖啡店里堆叠起来的盘子，放上新盘子和取用盘子都在上面[注14]。数组中最右侧的便是最后一个元素，也就是拥有最大索引值的那个元素。因为我们常常要对这个元素进行操作，所以Perl提供了专门的函数。

pop操作符就是其中一个，它负责取出数组中最后一个元素并将其作为返回值返回：

```
@array  = 5..9;
$fred   = pop(@array);  # $fred 变成9, @array 现在是 (5, 6, 7, 8)
$barney = pop @array;   # $barney 变成8, @array 现在是 (5, 6, 7)
pop @array;             # @array 现在是(5, 6)。(7被抛弃了。)
```

最后一行是在空上下文（void context）中使用pop操作符。所谓的“空上下文”只不过是表示返回值无处可去的一种说辞。这其实也是pop操作符常见的一种用法，用来删除数组中的最后一个元素。

如果数组是空的，pop什么也不做（因为没有任何元素可供移出），直接返回undef。

你也许已经注意到了，pop后面加不加括号都可以。这是Perl的惯例之一：只要不会因为拿掉括号而改变原意，括号就是可省略的[注15]。与此对应的是push操作符，用于添加一个元素（或是一串元素）到数组的尾端：

```
push(@array, 0);       # @array现在是 (5, 6, 0)
push @array, 8;        # @array现在是 (5, 6, 0, 8)
push @array, 1..10;    # @array得到了10个新元素
@others = qw/ 9 0 2 1 0 /;
push @array, @others; # @array又得到了5个新元素（共19个）
```

注13：　当然，我们是开玩笑的。但这个玩笑里隐含着一个事实：Perl并不擅长使用索引值来访问数组。如果使用pop、push或别的不需要索引值的操作方式，通常会比用了很多索引操作的程序更快，同时也能避免“大小差一（off-by-one）”错误，这通常也称为“栅栏柱（fencepost）”错误。时常会有 Perl 初学者想要比较Perl和C在速度上的差异，于是直接拿一个针对 C 语言优化过的排序算法（里面用到了很多索引操作）以Perl改写（一样也用到了很多索引操作）来进行性能比较后，很纳闷Perl为何这么慢。答案就是，拿Stradivari（斯特拉迪瓦里家族）制造的小提琴去敲铁钉，实在不应该被当成一种发音技法。

注14：　另外也常把数组当作队列（*queue*）来用，从一头加入，从另一头取出。

注15：　你可能会发现这句话其实是同义反复（tautology）。

注意，push的第一个参数或者pop的唯一参数都必须是要操作的数组变量——对列表直接量进行压入（push）或弹出（pop）操作是没有意义的。

shift和unshift操作符

push和pop操作符处理的是数组的尾端（或者说是数组的“右”边，最高下标值的部分，怎么理解都行）；相似地，unshift和shift操作符则是对数组的“开头”（或者说是数组的“左”边，最低下标值的部分）进行相应的处理。来看几个例子：

```
@array = qw# dino fred barney #;
$m = shift(@array);      # $m变成 "dino", @array现在是 ("fred", "barney")
$n = shift @array;       # $n变成 "fred", @array现在是 ("barney")
shift @array;            # 现在@array变空了
$o = shift @array;       # $o变成undef, @array还是空的
unshift(@array, 5);      # @array现在仅包含只有一个元素的列表 (5)
unshift @array, 4;       # @array现在是 (4, 5)
@others = 1..3;
unshift @array, @others; # @array 又变成了 (1, 2, 3, 4, 5)
```

与pop类似，对于一个空的数组变量，shift会返回undef。

splice操作符

push-pop和shift-unshift操作符都是针对数组首尾操作的，那么要是希望添加或移除数组中间的某些元素，又该怎么办呢？这正是splice操作符要做的事情。它最多可接受4个参数，最后两个是可选参数。第一个参数当然是要操作的目标数组，第二个参数是要操作的一组元素的开始位置。如果仅仅给出这两项参数，Perl会把从给定位置开始一直到末尾的全部元素取出来并返回：

```
@array = qw( pebbles dino fred barney betty );
@removed = splice @array, 2; # 在原来的数组中删掉fred及其后的元素
                             # @removed变成qw(fred barney betty)
                             # 而原先的@array则变成qw(pebbles dino)
```

我们可以通过第三个参数指定要操作的元素长度。请再读一遍这句话，很多人想当然以为第三个参数是结束位置，但实际并非如此，它表示要操作的元素个数，亦即长度。通过这个参数，我们就可以删掉数组中间的一个片段：

```
@array = qw( pebbles dino fred barney betty );
@removed = splice @array, 1, 2; # 删除dino和fred这两个元素
                                # @removed变成qw(dino fred)
                                # 而@array则变成qw(pebbles barney betty)
```

第四个参数是要替换的列表。之前我们看到的都是如何从现有的数组拿走元素，而其实

你也可以补充新元素到原来的数组中。替换的列表的长度并不一定要和拿走的元素片段一样长：

```
@array = qw( pebbles dino fred barney betty );
@removed = splice @array, 1, 2, qw(wilma); # 删除 dino 和 fred
                               # @removed 变成 qw(dino fred)
                               # @array 变成 qw(pebbles wilma
                               #                barney betty)
```

实际上，添加元素列表并不需要预先删除某些元素，把表示长度的第三个参数设为0，即可不加删除地插入新列表：

```
@array = qw( pebbles dino fred barney betty );
@removed = splice @array, 1, 0, qw(wilma); # 什么元素都不删
                               # @removed 变为 qw()
                               # @array 变为 qw(pebbles wilma dino
                               #                fred barney betty)
```

注意，wilma出现在dino之前的位置上。Perl从索引位置1的地方插入新列表，然后顺移原来的元素。

可能splice看起来并不起眼，但在其他语言中，要实现相同功能并不轻松。许多人为了达到相同目的，使用各种复杂概念和技术，比如链表什么的，但这无疑是把程序员的时间浪费在低层次的数据处理上，既不合理也不高明。

字符串中的数组内插

和标量一样，数组的内容同样可以被内插到双引号串中。内插时，会在数组的各个元素之间自动添加分隔用的空格[注16]：

```
@rocks = qw{ flintstone slate rubble };
print "quartz @rocks limestone\n";  # 打印5种以空格隔开的石头名
```

数组被内插后，首尾都不会增添额外空格；若你真的需要，自己动手加吧：

```
print "Three rocks are: @rocks.\n";
print "There's nothing in the parens (@empty) here.\n";
```

要是你忘记了数组内插是这样写的，那么当你把电子邮件地址放进双引号内时，可能会大吃一惊：

```
$email = "fred@bedrock.edu";  # 错！这样会内插 @bedrock 这个数组
```

注16：　事实上，此分隔符是由特殊变量$"的值指定的，默认为空格。

尽管我们只是想要显示电子邮件地址，Perl却看到了一个叫做@bedrock的数组，并继而尝试将之内插。对于某些版本的Perl，我们会看到这样的警告信息[注17]：

```
Possible unintended interpolation of @bedrock
```

要规避这种问题，要么将@转义，要么直接用单引号来定义字符串：

```
$email = "fred\@bedrock.edu"; # 正确
$email = 'fred@bedrock.edu';  # 另一种写法，效果相同
```

内插数组中的某个元素时，会被替换成该元素的值，正如你所愿：

```
@fred = qw(hello dolly);
$y = 2;
$x = "This is $fred[1]'s place";    # 得"This is dolly's place"
$x = "This is $fred[$y-1]'s place"; # 效果同上
```

请注意，索引表达式（index expression）会被当成普通字符串表达式处理。该表达式中的变量不会被内插。也就是说，假如$y包含字符串"2*4"，索引结果仍然为1，而非7。因为"2*4"被看作数字时（$y用于数值表达式中）相当于数字2[注18]。如果要在某个标量变量后面接着写左方括号，你需要先将这个方括号隔开，它才不至于被识别为数组引用的一部分。做法如下：

```
@fred = qw(eating rocks is wrong);
$fred = "right";               # 我们想要说"this is right[3]"
print "this is $fred[3]\n";    # 用到了$fred[3]，打印"wrong"
print "this is ${fred}[3]\n";  # 打印"right" （用花括号避免歧义）
print "this is $fred"."[3]\n"; # 还是打印right（用分开的字符串）
print "this is $fred\[3]\n";   # 还是打印right（用反斜线避免歧义）
```

foreach控制结构

如果能对整个数组或列表进行处理将是非常方便的，为此Perl提供了另一种控制结构。foreach（循环）能逐项遍历列表中的值，依次迭代（循环过程）：

```
foreach $rock (qw/ bedrock slate lava /) {
    print "One rock is $rock.\n"; # 依次打印所有三种石头的名称
}
```

注17：　某些5.6之前的Perl版本确实会将此视为致命错误，但后来改为警告信息，否则就太吵人了。不过在新版Perl当中，这已经不成问题。

注18：　当然，如果你启用了警告信息，Perl可能会提醒你"2*4"这个数字看起来怪怪的。

每次循环迭代时，控制变量（control variable，即此例中的$rock）都会从列表中取得新的值。第一次执行时，控制变量的值是"bedrock"；而第三次时，控制变量的值是"lava"。

控制变量并不是列表元素的复制品——实际上，它就是列表元素本身。也就是说，假如在循环中修改了控制变量的值，也就同时修改了这个列表元素，如同下面的代码片段所示。这种设计很有效，也被广泛认可，但是如果你没有预备，它却可以让你大吃一惊：

```
@rocks = qw/ bedrock slate lava /;
foreach $rock (@rocks) {
    $rock = "\t$rock";       # 在@rocks的每个元素前加上制表符
    $rock .= "\n";           # 同时在末尾加上换行符
}
print "The rocks are:\n", @rocks; # 各自占一行，并使用缩排
```

当循环结束后，控制变量的值会变成什么？它仍然是循环执行之前的值。Perl会自动存储foreach循环的控制变量并在循环结束之后还原。在循环执行期间，我们无法访问或改变已存储的值，所以当循环结束时，变量仍然保持循环前的值；如果它之前从未被赋值，那就仍然是undef：

```
$rock = 'shale';
@rocks = qw/ bedrock slate lava /;

foreach $rock (@rocks) {
    ...
}

print "rock is still $rock\n"; # 打印 "rock is still shale"
```

也就是说，假如你想将循环的控制变量取名为$rock的话，不必担心之前是否用过同名的变量。在随后第四章中介绍了子程序之后，我们会提供更好的处理方法。

Perl最喜欢用的默认变量：$_

假如在foreach循环开头省略控制变量，Perl就会用它最喜欢用的默认变量$_。这个变量除了名称比较特别以外，和其他标量变量（几乎）没什么差别。例如：

```
foreach (1..10) {  # 默认会用 $_ 作为控制变量
    print "I can count to $_!\n";
}
```

虽然它并非Perl中唯一的默认变量，却是最常用的一个。我们以后还会看到，在许多种情况下，当未告知Perl使用哪个变量或数值时，Perl都会自动使用$_，从而使程序员免于命名和键入新变量的痛苦。没错，这里的print就是一个例子，在没有参数时，它会打印$_的值：

```
$_ = "Yabba dabba doo\n";
print;  # 默认打印 $_ 变量的值
```

reverse操作符

`reverse`操作符会读取列表的值（也可能来自数组），并按相反的次序返回该列表。因此，假如你对范围操作符（`..`）只能递增计数感到失望，可以这样来弥补：

```
@fred   = 6..10;
@barney = reverse(@fred);  # 得10, 9, 8, 7, 6
@wilma  = reverse 6..10;   # 同上，但不需要额外的数组
@fred   = reverse @fred;   # 将逆序后的结果放回原来那个数组
```

值得注意的是，最后一行用了两次`@fred`。Perl总是会先计算（等号右边）要赋的值，再实际进行赋值操作。

请记住，`reverse`会返回次序相反的列表，但它并不会修改传进来的参数。假如返回值无处可去，那这种操作也就变得毫无意义：

```
reverse @fred;         # 错误——这不会修改@fred的值
@fred = reverse @fred; # 这才正确
```

sort操作符

`sort`操作符会读取列表的值（可也能来自数组），而且会根据内部的字符编码顺序对它们进行排序。对字符串而言，就是字符在计算机内部表示的代码点[注19]。在以往Unicode还没被Perl完整支持的年代，所谓的排序规则就是看ASCII码的大小，而现在Unicode不光是保留ASCII中的编码顺序，还为其他各式各样的字符设定代码点。字符代码点的顺序相当奇怪：大写字符排在小写字符前面，数字排在字母之前，而标点符号则散落各处。不管是否合乎逻辑，按照代码点大小排序只是默认做法。在第十四章里，我们将会看到如何按自定规则进行排序。`sort`操作符的参数应该是某个输入列表，然后对它排序，继而返回排序后的列表。来看几个例子：

```
@rocks   = qw/ bedrock slate rubble granite /;
@sorted  = sort(@rocks);        # 得bedrock, granite, rubble, slate
@back    = reverse sort @rocks; # 逆序后从slate到bedrock排列
@rocks   = sort @rocks;         # 将排序后的结果存回至@rocks
@numbers = sort 97..102;        # 得100, 101, 102, 97, 98, 99
```

从最后一个例子可以看出，将数字当成字符串来排序，这样的结果会不太对。根据默认的排序规则，任何以1开头的字符串会被排在以9开头的字符串之前。另外，它和

注19：默认的Unicode字符串排序不考虑本地化（locale）设置，但有时为了满足特定需要，你得自己打开本地化设置。

reverse一样不会修改参数，所以要对数组排序时，你必须将排序后的结果存回数组：

```
sort @rocks;           # 错误，这么做不会修改@rocks
@rocks = sort @rocks; # 现在收集来的石头排得井井有条
```

each操作符

从Perl 5.12版开始，已经可以针对数组使用each操作符了。但在此之前，each只能用于提取哈希的键-值对，具体细节我们留到第五章再讲。

每次对数组调用each，会返回数组中下一个元素所对应的两个值——该元素的索引以及该元素的值：

```
use 5.012;

my @rocks = qw/ bedrock slate rubble granite /;
while( my( $index, $value ) = each @rocks ) {
    say "$index: $value";
}
```

如果不用each来实现，就得自己根据索引从小到大依次遍历，然后借助索引值取得元素的值：

```
@rocks   = qw/ bedrock slate rubble granite /;
foreach $index ( 0 .. $#rocks ) {
    print "$index: $rocks[$index]\n";
}
```

实际上哪一个更方便得看你的具体需求，并无绝对定论。

标量上下文与列表上下文

这是本章最重要的一节，事实上，这甚至也是本书最重要的一节。哪怕说你的Perl水平完全取决于对本节的了解程度，也一点都不夸张。所以，假如你到目前为止一直都是随意翻阅本书的话，现在应该是全神贯注的时候了。

这并不是说本节有多么难懂。这里的概念其实非常简单：同一个表达式出现在不同的地方会有不同的意义。你应该不会陌生，因为在自然语言里，这种情况随处可见。以英语为例[注20]，假如有人问你单词“read”[注21]代表什么意思，你一定很难简单回答，因为

注20：　假如英语不是你的母语，这样的类比对你可能并不明显。但是每种语言都会有上下文感知（context sensitivity），因此你可以想出你自己的语言里的例子。

注21：　如果是在讲话而不是写书，那他们也许是在问单词“red”代表什么意思（译注：英文里read的过去式与red的发音完全相同。）。所以这么说无论如何都会造成混淆。就像Douglas Hofstadter 所说的，没有任何语言能够毫无歧义地表达思想，尤其是英语。

用在不一样的地方，表达的意思可能会不同。除非你知道上下文（*context*），否则一定没办法确认它的准确含义。

所谓上下文，指的是你如何使用表达式。实际上你已经看到过许多针对数字和字符串上下文的操作了。比如按照数字方式进行操作时得到的就是数字结果，而按照字符串方式进行某些操作时返回的则是字符串结果。并且，起到决定性因素的是操作符，而不是被操作的各种变量或直接量。2*3中的*作为对数字的乘法运算符号时得到的结果就是数字6，而<2 × 3>中的x则表示字符串重复操作，所以得到的结果是字符串222。这就是上下文在起作用。

当Perl在解析表达式时，你要么希望它返回一个标量，要么希望它返回一个列表[注22]。表达式所在的位置，Perl期望得到什么，那就是该表达式的上下文[注23]：

```
42 + something # 这里的something必须是标量
sort something # 这里的something必须是列表
```

就算*something*这个单词的拼写保持不变，它却会在某种情况下得出单一的标量值，而在另外的情况下产生一个列表[注24]。在Perl中，表达式总是根据所需要的上下文返回对应的值。以数组的“名称”[注25]为例：在列表上下文中，它会返回元素的列表；在标量上下文中，则返回数组中的元素个数：

```
@people = qw( fred barney betty );
@sorted = sort @people;  # 列表上下文：barney, betty, fred
$number = 42 + @people;  # 标量上下文：42 + 3 ，得45
```

即使是普通的赋值运算（对标量或列表赋值），都可以有不同的上下文：

```
@list = @people; # 得到3个人的姓名列表
$n = @people;    # 得到人数3
```

但请不要立刻得出结论，认为在标量上下文中一定会得到（列表上下文中返回的列表

注22：当然，Perl 也可能期望得到其他的东西。另有一些上下文在这里无法介绍。事实上，没有人知道 Perl 到底有多少种上下文，Perl长老团还没有对这个问题达成共识。

注23：这和我们人类使用的语言也有相似之处。如果我犯了一个文法上的错误，而你能马上注意到错误，那是因为你会在某些位置上希望有某些相关的用字。到后来，你一定也能以这种方法来读 Perl，不过在开始时得先想一想。

注24：当然，列表可能恰好只有一个元素，它也可能是空的，或具有任何数目的元素。

注25：对了，数组@people的名称就只是people。@符号只是一个限定符（qualifier）。

的）元素个数。有许多能产生列表的表达式[注26]所返回的东西可能比你想象中的还要丰富有趣。

不光如此，从我们积累的经验来看，仅仅通过对表达式形式上的判断是无法归纳出一个通用法则来的。每一个表达式都可能有它自己的特定规则。所以，实际上能够概括的规则也就是：哪种上下文更有意义，就应该是哪种上下文。但其实记住这条规则并没什么用。Perl是一种会尽量帮你完成常见任务的语言，往往它的选择就是你想要的结果。

在标量上下文中使用产生列表的表达式

有些表达式通常是用来生成列表的，假如在标量上下文中使用，结果会怎样？那就要看该表达式的作者怎么说了。基本上，这位作者就是Larry，他会在说明文档里详细解释。事实上，学习Perl的大部分时间都是在学习Larry的思维方式[注27]。因此，只要你能以Larry的方式思考，就可以预测Perl接下来会怎么做。但在你学习的过程中，可能还是需要看看Perl的说明文档。

某些表达式不会在标量上下文中返回任何值。比如sort在标量上下文中会返回什么？实际上没人需要统计列表排序后的元素个数。所以，除非有人修改了实现方式，否则sort在标量上下文中总是返回undef。

另一个例子是reverse。在列表上下文中，它很自然地返回逆序后的列表；在标量上下文中，则返回逆序后的字符串（先将列表中所有字符串全部连接到一起，再对结果中的每一个字符作逆序处理）[注28]：

```
@backwards = reverse qw/ yabba dabba doo /;
    # 会得到 doo, dabba, yabba
$backwards = reverse qw/ yabba dabba doo /;
    # 会得到 oodabbadabbay
```

注26：暂且不管这一小节的论点为何，其实“产生列表”的表达式和“产生标量”的表达式之间并无不同，任何表达式都可以产生列表或标量，根据上下文而定。因此当我们说“产生列表的表达式”时，我们指的是该表达式通常被用在列表上下文中。所以当它们不小心被用在标量上下文中时（像reverse或是@fred），你可能会因此感到惊讶。

注27：这其实也很自然，因为在创造Perl时，Larry已经试着揣测你的想法，并预测你的需求了！

注28：我们中的一个曾把Larry逼在电梯角落，请他解释解释这到底算要解决哪门子问题，但他好像要退到角落深处一般，怯生生地说道：“当时觉得这种做法很酷，所以就……”

刚开始时，往往很难一眼看出某个表达式到底是在标量上下文还是在列表上下文中。但请相信我们，这种直觉终将成为你的第二天性。

作为起步，下面列出一些常见的上下文：

```
$fred = something;            # 标量上下文
@pebbles = something;         # 列表上下文
($wilma, $betty) = something; # 列表上下文
($dino) = something;          # 还是列表上下文!
```

不要被只有一个元素的列表给蒙骗了，最后一个例子是列表上下文而不是标量上下文中。这里的括号非常重要，使得第四行和第一行完全不同。如果是给列表赋值（不管其中元素的个数），那就是列表上下文；如果是给数组赋值，那还是列表上下文。

下面列出的是之前看到过的表达式以及各表达式的上下文，我们分成两组，第一组是标量上下文中使用*something*的例子：

```
$fred = something;
$fred[3] = something;
123 + something
something + 654
if (something) { ... }
while (something) { ... }
$fred[something] = something;
```

第二组是列表上下文中的例子：

```
@fred = something;
($fred, $barney) = something;
($fred) = something;
push @fred, something;
foreach $fred (something) { ... }
sort something
reverse something
print something
```

在列表上下文中使用产生标量的表达式

这种情况十分简单：如果表达式求值结果为标量值，则自动产生一个仅含此标量值的列表：

```
@fred = 6 * 7; # 得到仅有单个元素的列表 (42)
@barney = "hello" . ' ' . "world";
```

好吧，其实这里有个小小的陷阱：

```
@wilma = undef; # 糟糕！结果是得到一个列表，而且仅有的一个元素为未定义的 (undef)
    # 这和下面的做法的效果完全不同：
@betty = ( );     # 这才是正确的清空数组的方法
```

因为undef是标量值，所以将undef赋值给数组并不会清空该数组。要清空的话，直接赋予一个空列表就可以了[注29]。

强制指定标量上下文

偶尔，在Perl想要列表上下文的地方你想要强制引入标量上下文，可以使用伪函数scalar。它可不是真正的函数，只不过告诉Perl这里要切换到标量上下文：

```
@rocks = qw( talc quartz jade obsidian );
print "How many rocks do you have?\n";
print "I have ", @rocks, " rocks!\n";         # 错误，这会输出各种石头的名称
print "I have ", scalar @rocks, " rocks!\n"; # 对了，打印出来的是石头种数
```

说来也奇怪，没有相应的函数可用来强行切入列表上下文。原因很简单，因为你根本用不到，相信我们。

列表上下文中的<STDIN>

我们之前看过的<STDIN>操作符在列表上下文中会返回不同的值。就像前面提过的，<STDIN>在标量上下文中会返回输入数据的下一行；在列表上下文中，则会返回所有剩下的行，直到文件结尾为止。返回的每一行都会成为列表中的元素。例如：

```
@lines = <STDIN>; # 在列表上下文中读取标准输入
```

当输入数据来自某个文件时，它会读取文件的剩余部分。但如果输入数据的来源是键盘，应该如何发送文件结尾标记呢？对Unix或类似系统（包括Linux和Mac OS X）来说，通常可以键入Control+D[注30]来告知系统，不会再有任何输入了。即使这个特殊字符会被打印在屏幕上，Perl也不会看到它[注31]。对DOS/Windows系统来说，就要用

注29：好吧，在真实世界的算法中，如果变量是在某个特定词法作用域内声明的，就不用显式清空。因为程序运行到此范围外时就会自动恢复初始的清空状态。所以这种写法在实际程序中并不多见。有关词法作用域的内容，我们将在第四章中介绍。

注30：这只是默认按键，你可用stty命令加以变更。但是这个默认配置很普遍——我们还没见过哪种Unix系统会使用其他字符来代表文件结尾。

注31：实际上是操作系统“看到了”这个控制按键，并据此向应用程序报告“文件结尾”的信号。

Control+Z了[注32]。假如你使用的是其他操作系统，请查看系统的说明文档或就近找专家咨询。

假如运行程序的人在键入了三行数据之后，又用适当的按键来表示文件结尾，最后得到的数组就会包含三个元素。每个元素都是一个以换行符结尾的字符串，因为这些换行符也是输入的内容。

如果在读取这些数据行时，能一次性chomp所有的换行符岂不更好？可以直接把数组交给chomp，它会自动去掉每个元素的换行符。例如：

```
@lines = <STDIN>; # Read all the lines
chomp(@lines);    # 去掉所有换行符
```

不过更常见的做法是按之前介绍的风格写成：

```
chomp(@lines = <STDIN>); # 读入所有行，换行符除外
```

虽然你可以按个人喜好来决定怎么做，不过大多数Perl程序员都会倾向于第二种更紧凑的写法。

输入数据一经读取，就无法回头再读一次。这也许对你（但不见得对所有人）而言是理所当然的事[注33]，一旦读到文件结尾，就没有更多输入数据可供读取了。

如果要读入的是某个400MB大小的日志文件，会有什么结果？“行输入”操作符会读取所有的数据行，同时占用大量内存[注34]。Perl不会限制你做这种事，但是系统中的其他用户（当然还有系统管理员）很可能会抗议。在读取大量数据时，通常应该考虑替代方案，避免一次将全部数据读进内存。

注32：在 DOS/Windows上的某些Perl版本有个缺陷：按下Control+Z之后，在屏幕上输出的第一行会被盖掉。在这些系统上，你可以在读进输入后输出一个空白行（即"\n"）来解决这个问题。

注33：没错，如果你的输入来源可以使用seek函数，就可以定位到起点重新读入。但这种做法并不适合在此处讨论。

注34：通常，需要的内存远超过文件的大小。换句话说，一个400MB大小的文件在被读进数组时通常会占据至少1GB的内存。因为Perl会分配富裕的内存来节省事后的操作时间。这是个不错的取舍：假如内存不够，还可以再买一些；但如果时间不够，那就完了。

习题

以下习题答案参见第310页上的“第三章习题解答”一节：

1. [6]写一个程序，读入一些字符串（每行一个），直到文件结尾为止。然后，再以相反顺序输出这个列表。假如输入来自键盘，你需要在Unix系统上键入Control+D或在Windows系统上键入Control+Z来表示输入的结束。

2. [12]写一个程序，读入一些数字（每行一个），直到文件结尾为止。然后，根据每一个数字输出如下名单中相应的人名（请将这份名单写到程序里，也就是说，你的程序代码里应该出现这份名单）。比方说，如果输入的数字是1、2、4和2，那么输出的人名将会是fred、betty、dino和betty：

   ```
   fred betty barney dino wilma pebbles bamm-bamm
   ```

3. [8]写一个程序，读入一些字符串（每行一个），直到文件结尾为止。然后，请按照ASCII码顺序输出所有字符串。换句话说，假如你键入的是fred、barney、wilma、betty，输出应该显示barney betty fred wilma。所有的字符串可以成一行输出吗？或者分开在不同行输出？你能分别让程序以这两种方式输出吗？

第四章

子程序

我们已见过并用过一些内置的系统函数，像chomp、reverse和print等。但是，就如同其他语言一样，Perl也可以让你创建子程序（*subroutine*），也就是用户自定义的函数[注1]。它让我们可以重复利用已有的代码。子程序的名称也属于Perl标识符的范畴（即由字母、数字和下划线组成，但不能以数字开头），有时候视情况会以“与号”（&）开头。关于何时可以省略以及何时必须加上这个放在标识符前面的与号，我们将在本章末尾进行介绍。若无其他声明，我们都将使用该符号，这通常都是比较保险的做法。当然，也有一定不能用与号的情形，届时我们自然会作说明。

子程序名属于独立的名字空间，这样Perl就不会将同一段代码中的子程序&fred和标量$fred搞混——虽然实际编程时没人会为两种不同的东西取一样的名字。

定义子程序

要定义你自己的子程序，可使用关键字sub、子程序名（不包含与号）以及用花括号封闭起来的代码块，这部分代码就是子程序的主体。例如：

```
sub marine {
    $n += 1;  # 全局变量 $n
    print "Hello, sailor number $n!\n";
}
```

注1：　在Perl中，我们一般不会像Pascal程序员那样区分有返回值的函数（*function*）和无返回值的过程（*procedure*）。但请注意，子程序总是由用户定义的,而函数则不一定。所以，有时候“函数”这个词可能被当成“子程序”的同义词，但它也可能指某个Perl的内置函数。大多数时候，可以自己定义的是子程序，而不是内置函数，所以我们这章的标题是“子程序”。

子程序可以被定义在程序中的任意位置，有C语言或Pascal背景的程序员喜欢将子程序的定义放在文件的开头。也有人喜欢将它们放在文件的结尾，从而使程序主体出现在开头部分。你可以随意选择采用哪种风格。不管怎样，你都不需要对子程序进行事先声明 [注2] 。子程序的定义是全局的，除非你使用一些强有力的技巧，否则不存在所谓的私有子程序 [注3] 。假如你定义了两个重名的子程序 [注4] ，那么后面的那个子程序会覆盖掉前面的那个。如果启用了警告信息，Perl会告诉你有子程序被重复定义。一般来说，重名总归是不够妥当的做法，也会让程序维护员感到困惑。

正如你在之前的例子中看到的，你可以在子程序中使用任何全局变量。事实上,目前你见过的所有变量都是全局的，这意味着你可以在你的程序的任何位置访问这些变量。很多语言学的好事者又该为此感到“震惊”了，但Perl开发组很多年前就组织了一队手持“火把”的“愤青团”，把这些好事者“赶出城”。我们稍后会在第78页的“子程序中的私有变量”一节中学到如何建立私有变量。

调用子程序

你可以在任意表达式中使用子程序名（前面加上与号）来调用它 [注5] ：

```
&marine;  # 打印 Hello, sailor number 1!
&marine;  # 打印 Hello, sailor number 2!
&marine;  # 打印 Hello, sailor number 3!
&marine;  # 打印 Hello, sailor number 4!
```

通常，我们把对子程序的调用称为呼叫（*calling*）子程序。除了上面例子中的用法以外，你还会在本章后面看到其他调用子程序的用法。

注2： 除非你的子程序比较特别并且声明了“原型（prototype）”。原型是用来告诉编译器如何解析和解释传进来的参数的。但现在这种用法已经非常少见——详细信息请参考*perlsub*说明文档。

注3： 如果你想了解这些强大的技巧，请阅读Perl的文档中关于保存在私有（lexical）变量中的代码引用（coderefs）的相关章节。

注4： 我们不会深入讨论不同包内的同名子程序，但我们的另一本书《Intermediate Perl》会有说明。

注5： 通常还会在后面加上一对括号，哪怕是空括号。因为子程序会继承调用者的@_值，这个我们马上将会讨论，所以请继续阅读，不然你写的代码的效果可能会和你预期的不同！

返回值

子程序被调用时一定是作为表达式的某个部分，即使该表达式的求值结果不会被用到。之前我们在调用&marine时，先对包含调用动作的表达式求值，但随即就把结果丢弃了。

很多时候，我们需要调用某个子程序并对它的返回值作进一步的处理。所以我们需要注意子程序的返回值。在Perl中，所有的子程序都有一个返回值——子程序并没有“有返回值”或“没有返回值”之分。但并不是所有的Perl子程序都包含有用的返回值。

既然任何Perl子程序都有返回值，那么规定每次必须写“return”某值就显得非常费事，所以Larry将它简化了。在子程序的执行过程中，它会不断进行运算，而最后一次运算的结果（不管是什么）都会被自动当成子程序的返回值。

比如我们定义下面这个子程序，最后一个是加法表达式：

```
sub sum_of_fred_and_barney {
    print "Hey, you called the sum_of_fred_and_barney subroutine!\n";
    $fred + $barney;  # 这就是返回值
}
```

这个子程序里最后执行的表达式就是计算$fred与$barney的总和。因此，$fred与$barney的总和就是返回值。以下是实际运行的情况：

```
$fred   = 3;
$barney = 4;
$wilma  = &sum_of_fred_and_barney;      # $wilma 为 7
print "\$wilma is $wilma.\n";

$betty  = 3 * &sum_of_fred_and_barney;  # $betty 为 21
print "\$betty is $betty.\n";
```

这段代码会输出以下内容：

```
Hey, you called the sum_of_fred_and_barney subroutine!
$wilma is 7.
Hey, you called the sum_of_fred_and_barney subroutine!
$betty is 21.
```

此处的print语句只用于协助调试，让我们得以确定该子程序被调用到了，程序完工后便可将它删除。不过，假设你在这段程序代码的结尾新增一条print语句，像这样：

```
sub sum_of_fred_and_barney {
    print "Hey, you called the sum_of_fred_and_barney subroutine!\n";
    $fred + $barney;  # 这不是返回值!
    print "Hey, I'm returning a value now!\n";      # 糟糕!
}
```

最后执行的表达式并非加法运算，而是print语句。它的返回值通常是1，表示“成功输出信息”[注6]，但它不是我们真正想要返回的值。所以在子程序里增加额外的程序代码时，请小心检查最后执行的表达式是哪一个，确定它是你要的返回值。

那么，那个第二个（不完善的）子程序中$fred与$barney相加的结果怎样了？我们并没有将总和存储起来，所以Perl会丢弃它。启用警告信息时，Perl会注意到将两数相加的结果丢弃是毫无用处的，并显示“a useless use of addition in a void context”之类的信息来警告你。其中的术语“*void context*（空上下文）”表示运算结果没有存储到变量里，也未被任何函数使用。

“最后执行的表达式”的准确含义真的就是最后执行的表达式，而非程序代码的最后一行。比如下面这个子程序，它会返回$fred和$barney两者中值较大者：

```
sub larger_of_fred_or_barney {
    if ($fred > $barney) {
        $fred;
    } else {
        $barney;
    }
}
```

最后执行的表达式是自成一行的$fred或$barney，所以它们将充当子程序的返回值。必须等到执行阶段得知这些变量的内容后，我们才会知道返回值到底是$fred还是$barney。

目前为止都还只是些范例而已。接下来会介绍如何在每次调用子程序时传入不同的值，而不再依靠全局变量。使用传入参数才是我们最终的实际用法。

参数

假如子程序larger_of_fred_or_barney不强迫我们一定用全局变量$fred和$barney，那么使用上会更灵活。在目前的状况下，假如我们想从$wilma和$betty中取出较大值，必须先将它们复制到$fred和$barney，才可以使用子程序larger_of_fred_or_barney。此外，假如别处需要用到$fred和$barney原本的值，我们还得先将它们复制到其他变量，比方说$save_fred和$save_barney。然后当子程序执行完毕后，我们又必须将这些值复制回$fred和$barney。

还好，Perl子程序可以有参数（*argument*）。要传递参数列表到子程序里，只要在子程序调用的后面加上被括号圈引的列表表达式就行了。例如：

注6：　print语句返回的是执行结果，成功执行返回真，否则返回假。至于如何判断失败类型，我们会在第五章介绍。

```
$n = &max(10, 15);  # 包含两个参数的子程序调用
```

参数列表将会被传入子程序，让子程序随意使用。当然，得先将这个列表存在某处，Perl会自动将参数列表化名为特殊的数组变量@_，该变量在子程序执行期间有效。子程序可以访问这个数组，以判断参数的个数以及参数的值。

这表示子程序的第一个参数存储于$_[0]，第二个参数存储于$_[1]，依此类推。但是，请特别注意，这些变量和$_变量毫无关联，就像$dino[3]（数组@dino中的元素之一）与$dino（一个独立的标量变量）毫无关联一样。参数列表总得存进某个数组变量里，好让子程序使用，而Perl将这个数组叫做@_，仅此而已。

现在，你可以写一个类似于&larger_of_fred_or_barney的子程序&max，但是可以使用子程序的第一个参数（$_[0]）而不用$fred，也可以使用子程序的第二个参数（$_[1]）而不用$barney。因此，最后你可以写成这样：

```
sub max {
    # 请比较它和子程序&larger_of_fred_or_barney的差异
    if ($_[0] > $_[1]) {
        $_[0];
    } else {
        $_[1];
    }
}
```

好吧，就像上面所说的，你可以那么写。但这里的一堆下标让程序变得不雅观，而且难以阅读、编写、检查和调试。不用担心，我们马上就会看到更好的办法。

这个子程序还存在另一个问题。&max这个名字虽然既好听又简洁，但却没有说明这个子程序只接受两个参数：

```
$n = &max(10, 15, 27);  # 糟糕!
```

多余的参数会被忽略——反正子程序也不会用到$_[2]，所以Perl并不在乎里面是否有值。参数如果不足也会被忽略——如果用到超出@_数组边界的参数，只会得到undef。本章稍后我们就会看到如何写出更好的&max子程序，并且让它可以配合任意数目的参数。

实际上，@_变量是子程序的私有变量[注7]。假如已经有了全局变量@_，则该变量在

注7：　除非调用子程序的时候前面加了与号，且后面没有跟括号（或者参数）。那种特殊情况下 @_ 数组会从调用者上下文中被继承下来。一般来说这不是个好主意，但偶尔也能派上用场。

子程序调用前会先被存起来，并在子程序返回时恢复原本的值[注8]。这也表示子程序可以将参数传给其他程序，而不用担心遗失自己的@_变量——就算是嵌套的子程序（nested subroutine）调用它自己的@_变量时也一样。即使子程序递归调用自己，每次调用的仍然是一个新的@_。所以在当前的子程序调用中，@_总是包含了它的参数列表。

子程序中的私有变量

既然每次调用子程序时Perl都会给我们新的@_，难道不能利用它构造私有变量吗？答案当然是可以。

默认情况下，Perl里面所有的变量都是全局变量，也就是说，在程序里的任何地方都可以访问它们。但你随时可以借助my操作符来创建私有变量，我们称之为词法变量（*lexical variable*）：

```
sub max {
    my($m, $n);        # 该语句块中的新私有变量
    ($m, $n) = @_;     # 将参数赋值给变量
    if ($m > $n) { $m } else { $n }
}
```

这些变量属于封闭语句块的私有变量（或者也可以称作有限作用域（*scoped*）变量），语句块之外任意地方的$m或$n都完全不受这两个私有变量的影响。反过来也是，外部变量同样无法影响内部的私有变量，存心也好意外也罢，两不相犯[注9]。所以，我们可以把这个子程序放进世界上任何一个Perl程序里，不用担心它和哪个程序中（可能存在）的$m和$n变量冲突[注10]。另外值得一提的是，在前一个例子中的if语句块中，作为返回值的表达式后面没有分号。虽然Perl允许你省略语句块中最后一个分号[注11]，但实际上通常只有像前面的例子那样，代码简单到整个语句块内只有一行时，才可以省略分号。

前一个例子中的子程序还可以进一步简化。你是否注意到列表（$m,$n）出现了两次？

注8：　你也许认识到了这和上一章里介绍过的foreach循环保存控制变量的机制相同，变量的值都是被Perl自动地保存和恢复的。

注9：　有经验的程序员应该知道，词法变量可以通过对它的引用，从其词法作用域外部被访问到，但绝不是通过变量名。有关引用的内容，我们在《Intermediate Perl》一书中有详细讲解。

注10：　当然，如果那个程序中碰巧也有个&max子程序，那就肯定会出乱子。

注11：　分号的实际作用仅仅是分隔语句，而不是必需的语句结束标记。

其实my操作符也可以应用到括号内的变量列表，所以习惯上会将这个子程序中的前两行语句合并起来：

```
my($m, $n) = @_;  # 对子程序的参数命名
```

这一行语句会创建私有变量并为它们赋值，让第一个参数的名称变成较好记的$m，而第二个参数的则为$n。几乎所有的子程序都会以类似的程序代码作为开头。当你看到这一行时就会知道，这个子程序具有两个标量参数，而在子程序内部，它们分别称为$m和$n。

变长参数列表

在真实的Perl代码中，常常把更长的（任意长度的）列表作为参数传给子程序。这延续了Perl“去除不必要的限制”的理念。当然，这和许多传统程序语言不一样，它们习惯于“强类型（strictly typed）”子程序，也就是说，只允许一定个数的参数并且预先限制它们的类型。Perl的灵活是件好事，不过当子程序以超乎作者预期个数的参数被调用时（例如我们之前看到的子程序&max），也许会造成问题。

当然，通过检查@_数组的长度其实也很容易确定参数个数是否正确。比方说，我们可以将&max写成下面这样以检查其参数列表[注12]：

```
sub max {
    if (@_ != 2) {
        print "WARNING! &max should get exactly two arguments!\n";
    }
    # 其余代码和前面一样……
    .
    .
    .
}
```

上面的if判断是在标量上下文中直接使用数组“名称”来取得数组元素的个数，你应该在第三章中见过这个用法。

但在实际编写的Perl程序中这种检查方式很少见，更好的做法是让子程序自动适应任意数目的参数。

改进的&max子程序

现在让我们来改写&max，使它能够接受任意数目的参数：

注12： 当你从第五章学了warn函数后，便可以换用它来输出合适的警告信息。如果出错情况比较严重，可以用die，这个函数同样会在那一章学到。

```
$maximum = &max(3, 5, 10, 4, 6);

sub max {
    my($max_so_far) = shift @_;  # 数组中的第一个值，暂时把它当成最大值
    foreach (@_) {               # 遍历数组@_中的其他元素
        if ($_ > $max_so_far) {  # 当前元素比$max_so_far更大吗？
            $max_so_far = $_;
        }
    }
    $max_so_far;
}
```

上面的程序代码使用了一般称为“高水线（high-watermark）”的算法：大水过后，在最后一波浪消退时，高水线会标示出所见过的最高水位。本例中，$max_so_far记录了高水线，所以最后$max_so_far变量中的值就是我们要找的最大值。

第一行程序代码会对参数数组@_进行shift操作并将所得到的3（范例程序里的第一个参数）存入$max_so_far变量。所以@_现在的内容为(5,10,4,6)，因为3已被移走。现在最大的数字是目前唯一见过的值：3，也就是第一个参数。

然后foreach循环会遍历参数列表@_里剩余的元素。循环的控制变量默认为$_（别忘了@_和$_没有任何关系，它们的名称相似纯属巧合）。循环第一次执行时，$_是5，而if进行比较时看到$_比$max_so_far还大，所以$max_so_far会被设成5——新的高水线。

第三次循环时，$_是10。这是新的最大值，所以它会被存入$max_so_far。

到第四次时，$_是4。这时if比较的结果为假，因为$_不比$max_so_far（即10）大，所以会跳过if里的程序代码。

最后，$_是6，因此if里的程序代码又被跳过一次。这是最后一次执行循环，所以整段循环就执行完了。

此时，$max_so_far就变成了我们的返回值。既然它是目前见过的最大值，并且我们已经遍历过所有数字，那它一定是列表中最大的值：10。

空参数列表

即使有超过两个的参数，改进后的&max算法仍然可以得出正确结果。但假如没有任何参数传入，又会发生什么事情呢？

乍听之下，这似乎有点杞人忧天。毕竟，怎么可能会有人调用&max却不传入任何参数呢？不过，也许有人会写出下面这样的代码：

```
$maximum = &max(@numbers);
```

数组@numbers有时可能只是一个空列表。比如，也许数组内容是程序从文件读入的，但文件却是空的。所以，这种情况下&max会如何处理？

子程序的第一行会对参数数组@_（现在是空的）进行shift操作，以此作为$max_so_far的值。这并不会出错，因为数组是空的，所以shift会返回undef给$max_so_far。

现在foreach循环要遍历@_数组，但由于它是空的，所以循环本身不会执行。

接下来，Perl会将$max_so_far的值undef作为子程序的返回值。从某种角度来看，那是正确的结果，因为在空列表中没有最大值。

当然，调用这个子程序的人得留意，返回值可能是undef——除非他能确保参数列表不为空。

关于词法（my）变量

事实上，词法变量可使用在任何语句块内，而不仅限于子程序的语句块。比如说，它可以在if、while或foreach的语句块里使用：

```
foreach (1..10) {
    my($square) = $_ * $_;  # 该循环中的私有变量
    print "$_ squared is $square.\n";
}
```

上面的例子中，变量$square对其所属语句块（也就是foreach循环的语句块）来说是私有的。如果变量的定义并未出现在任何语句块里，则该变量对于整个程序源文件来说就是私有的。到目前为止，你的程序还用不到两个以上的程序源文件[注13]，所以这还不成问题。这里的重点在于，词法变量的作用域（scope）受限于定义它的最内层语句块（或文件）。只有语句块上下文作用域内的程序代码才能以$square这个名称使用该变量。这为程序维护提供了便利。如果$square的值出错了，那么就可以在有限的源代码范围内找到原因。有经验的程序员都知道（这往往是付出惨痛代价换来的），将变量作用域圈定在一页或少数几行代码内，的确能加快开发及测试周期。

还需要注意的是，my操作符并不会更改变量赋值时的上下文：

```
my($num) = @_;  # 列表上下文，和 ($num) = @_; 相同
my $num  = @_;  # 标量上下文，和 $num = @_; 相同
```

在第一行里，按照列表上下文，$num会被设为第一个参数；在第二行里，按照标量上下

注13：　我们在《Intermediate Perl》一书中有谈到可供重复使用的库与模块的概念。

文，它会被设为参数的个数。这两行都有可能是程序员的本意，我们并不能单从一行程序代码断定你要的是哪个，因此在你搞错时，Perl无法提出警告（当然，你不会真的把这两行放在同一个子程序里，因为相同的作用域内不能定义两个同名的词法变量，以上只是举例而已）。所以，看到这样的程序代码时，你可以忽略my这个词，直接判断变量赋值时的上下文。

请记住，在my操作符不加括号时，只能用来声明单个词法变量[注14]：

```
my $fred, $barney;        # 错！没声明$barney
my($fred, $barney);       # 两个都声明了
```

当然，你也可以使用my来创建新的私有数组[注15]：

```
my @phone_number;
```

所有新变量的值一开始都是空的：标量被设为undef，数组被设为空列表。

在日常Perl编程当中，你最好对每个新变量都使用my声明，让它保持在自己所在的词法作用域内。在第三章中，你已经看到过如何在foreach循环中定义自己的控制变量，而这个控制变量也可以声明为词法变量：

```
foreach my $rock (qw/bedrock slate lava /) {
    print "One rock is $rock.\n";  # 依次输出每块石头的名字
}
```

这一点很重要，接下来的小节里，我们会全面使用这种声明变量的方式。

use strict编译指令

Perl是一门相当宽容的编程语言[注16]。但你也许希望Perl能更严格一些，多一点约束力。要达成这一点，不妨试试看use strict编译指令。

所谓编译指令（pragma），其实不过是提供给编译器的某些指示，告诉它如何处理接下来的代码。这里的use strict编译指令是要告诉Perl内部的编译器接下来（代码块或是程序源文件中）的代码应该稍加严谨一点，遵循一些优良的编程风格。

注14：　和往常一样，打开警告开关可以在滥用my的时候发出警告信息。而如果使用strict编译指令（稍后会提及）的话，就会在一开始就禁止滥用行为。

注15：　或者哈希也行，这个你将在第六章看到。

注16：　我猜你大概还没注意到。

这么做的重要性何在？嗯，设想你正在写程序并键入下面这行代码：

```
$bamm_bamm = 3;  # Perl 会自动创建这个变量
```

然后你继续写了一些代码。当上一行代码被新写的代码挤出屏幕顶端后，你又写了下面这行代码，想增加那个变量的值：

```
$bammbamm += 1;  # 糟了！
```

因为Perl看到了一个新的变量名（变量名中的下划线是有意义的），它会创建一个新的变量，然后增加它的值。如果你有先见之明，已经启用了警告功能，那么Perl就会警告你这两个（或其中之一）全局变量名在程序中只出现过一次。可是，如果你不够谨慎，那么这两个变量名都会在程序里并存，Perl也并不发出警告。

要告诉Perl你愿意接受更严格的限制，请将`use strict`这个编译指令放在程序开头（或者任何需要强制使用约束规则的语句块或文件内）：

```
use strict;  # 强制使用一些严格的、良好的编程风格
```

从Perl 5.12开始，如果使用编译指令指定最低Perl版本号的话，就相当于隐式打开约束指令：

```
use 5.012; # 自动加载strict编译指令
```

现在，所有的约束当中[注17]，Perl首先会坚持要求你声明所有新出现的变量。一般加上my就可以了[注18]：

```
my $bamm_bamm = 3;  # 新的词法变量
```

现在，如果你再像之前那样拼错变量名，Perl就会抗议，说你从未声明过$bammbamm这个变量，因此错误在编译阶段就会被找出来：

```
$bammbamm += 1;  # 无此变量：属于编译时致命错误
```

当然，此限制只适用于新创建的变量，Perl的内置变量（比如$_和@_）则完全不用事先

注17：要学习其他的限制规则，请阅读strict的说明文档。知道任何编译指令的名字也就知道了相应文档名，所以你可以使用*perldoc strict*（或者你的系统上的查看文档命令）来查看strict的文档。简单来说，大多数情况下其他的限制规则会要求对字符串使用引号，并且所有的引用必须是硬引用（我们不会在这里讨论关于引用的概念，不管是软饮用还是硬引用，但在《Intermediate Perl》一书中会有详细讲解）。放心，这些限制规则不会对Perl的初学者造成困扰。

注18：当然除此之外，还有别的声明方式。

声明[注19]。如果你在程序写完之后再加use strict，通常会得到一大堆警告信息，所以如果有需要，最好在开始写程序时就用它。

根据大部分人的建议，比整个屏幕长的程序都应该加上use strict。这一点我们也相当认同。

从现在开始，即使没有清楚写出use strict，我们所提供的大部分（但并非全部）范例程序都会在它的限制下执行。也就是说，在合适的位置尽可能用my来声明变量。不过，即使我们没有从头到尾都这样做，仍然建议你尽可能在程序里加进use strict。时间久了，你就会感激我们的。

return操作符

如果想在子程序执行到一半时停止执行，该怎么办呢？这正是return操作符要做的事，它会立即停止执行并从子程序内返回某个值：

```
my @names = qw/ fred barney betty dino wilma pebbles bamm-bamm /;
my $result = &which_element_is("dino", @names);

sub which_element_is {
    my($what, @array) = @_;
    foreach (0..$#array) {  # 数组@array中所有元素的索引
        if ($what eq $array[$_]) {
            return $_;         # 一发现就提前返回
        }
    }
    -1;                     # 没找到符合条件的元素（写不写return都没关系）
}
```

我们希望这个子程序能找出@names数组中值为dino的元素的索引。首先，用my声明参数名：一个是$what，表示要搜索的内容，另一个是@array，表示供搜索的目标数组。请看上面的代码，这个数组其实是@names的拷贝。foreach循环依次取出@array的索引值（第一个索引值为0，最后一个索引值为$#array，这个我们已经在第三章见到过了）。

每一次执行foreach循环时，我们会检查$what中的字符串是否等于[注20]位于当前索引

注19：　另外最好别声明$a和$b这两个变量，因为在某些情况下，它们默认被用作sort函数的内置变量，换用其他的变量名吧。事实上use strict不能限制这两个变量，而这居然也常被当作bug提交给Perl维护者。

注20：　这里用的是字符串相等比较符eq，而不是数值相等比较符==，你注意到这点了吗？

的数组@array的元素。如果两者相等，我们就立刻返回其索引值。这是关键字return最常见的用法：立即返回某个值，而不再执行子程序的其余部分。

但是，假如我们找不到符合条件的那个元素呢？本例中，子程序的作者选择返回-1作为“查无此值”的数字代号。虽然在这个例子里，返回undef可能会更符合Perl的风格（Perlish），但他还是决定用-1。最后一行写成return -1也行，但实际上return并不是必需的。

有些程序员在每次返回值时都会用return，以明确表明它是返回值。比如在子程序执行到中间部分就已经得出结论的话，就可以使用return关键字立即返回，就像本章之前提到&larger_of_fred_or_barney时说的那样。而到子程序末尾再返回结果的话，则完全可以省略return，当然写了也没什么坏处。只是很多Perl程序员都觉得那是完全多余的7次按键罢了。

省略与号

遵照先前的承诺，我们现在要告诉你，在调用子程序时何时可以省略与号。如果编译器在调用子程序前看到过子程序的定义，或者Perl通过语法规则判断它只能是子程序调用，那么对待该子程序就可以像内置函数那样，在调用时省略与号（不过我们马上就会看到，这个规则其实还潜藏了一个例外）。

换句话说，假如Perl单从语法上便能看出它是省略了与号的子程序调用，通常就不会有什么问题。也就是说，你只要将参数列表放进括号里，它就一定是函数[注21]调用：

```
my @cards = shuffle(@deck_of_cards);  # &shuffle 上的 & 是多余的
```

或者，如果Perl的内部编译器已经见过子程序的定义，那么通常也可以省略与号。并且，连参数列表两边的括号都可以省略：

```
sub division {
    $_[0] / $_[1];                   # 用第一个参数除以第二个参数
}

my $quotient = division 355, 113;  # 用的就是之前定义的 &division
```

上面的程序之所以能够运行，是因为它符合“加不加括号都不会产生歧义”的原则。

但不要把子程序定义放在调用语句的后面，否则编译器就无法提前判断division的意

注21：　在本例中，这个函数就是子程序&shuffle。但正如你将要看到的那样，它也可能是个内置函数。

义。编译器需要在子程序调用前先看到子程序定义，才有办法像对内置函数般调用该子程序；否则，编译器不知道该如何处理这个表达式。

不过这还不算之前提到的例外。真正的例外是：假如这个子程序和Perl内置函数同名，你就必须使用与号调用。原因很简单，也是为了避免歧义。加上与号，就说明调用的是你自己定义的子程序。所以只能在没有同名内置函数的情况下省略与号：

```
sub chomp {
    print "Munch, munch!\n";
}

&chomp;  # 这里必须使用 &，绝不能省略!
```

如果少了与号，就算之前定义过子程序&chomp，实际仍然会调用内置函数chomp。所以，真正的省略规则如下：除非你知道Perl所有的内置函数名，否则请务必在调用函数时使用与号。这意味着，你最初写的100个程序应该始终加上与号。但若看到别人的程序中省略与号的写法，那未必是错的，也许他很清楚Perl没有这个名字的内置函数[注22]。当程序员打算以调用Perl内置函数的方式来调用自己的子程序时，通常是在编写模块（*module*）的时候，他们经常使用原型（*prototype*）来告诉Perl该如何对待子程序的参数。构造模块是高级主题，如果你已经预备好了，可以参阅Perl的说明文档（特别是*perlmod*和*perlsub*文档），以便了解更多关于子程序原型和构造模块的信息[注23]。

非标量返回值

子程序不仅可以返回标量值，如果你在列表上下文调用它[注24]，它还能返回列表值。

假设你想取出某段范围的数字（如同范围操作符..的递增序列），只是你不但想递增，还想递减序列。虽然范围操作符只能产生递增序列，不过要反过来取也不是什么难事：

```
sub list_from_fred_to_barney {
    if ($fred < $barney) {
        # 从 $fred 数到 $barney
        $fred..$barney;
    } else {
```

注22：当然，也可能的确是个错误。究竟如何，不妨打开*perlfunc*和*perlop*文档看一看是否有同名的内置函数。另外，如果打开了警告开关，Perl会给你相关的警告信息。

注23：或者，进阶阅读《Intermediate Perl》一书。

注24：你可以用wantarray函数来判断子程序是在标量上下文还是列表上下文环境中执行，它可以让你写的子程序更加智能，看外面的需要返回特定的标量值或列表值。

```
        # 从 $fred 倒数回 $barney
        reverse $barney..$fred;
    }
}

$fred = 11;
$barney = 6;
@c = &list_from_fred_to_barney; # @c 的值为 (11, 10, 9, 8, 7, 6)
```

此例中，我们会先用范围操作符取得从6到11的列表，再用reverse操作符把它反转过来。最后的结果就是我们想要的：从$fred(11)倒数到$barney(6)的列表。

其实你还可以什么都不返回。单写一个return不给任何参数时，在标量上下文中的返回值就是undef，在列表上下文中则是空列表。这通常用于表示子程序执行有误，它告诉调用者无法取得有意义的返回值。

持久性私有变量

在子程序中可以使用my操作符来创建私有变量，但每次调用这个子程序的时候，这个私有变量都会被重新定义。而使用state操作符来声明变量，我们便可以在子程序的多次调用期间保留变量之前的值，并将变量的作用域局限于子程序内部。

回到本章第一个例子，我们有个名为marine的子程序，它会使全局变量的值每次递增：

```
sub marine {
    $n += 1;  # 全局变量$n
    print "Hello, sailor number $n!\n";
}
```

既然刚介绍过strict，不妨把它加到这段程序中。很快你就会发现，全局变量$n从未被声明，Perl立即提示错误。此外，我们也不能用my声明$n为词法变量，因为词法变量无法存续，语句块一结束，它的值就会被抛弃。

我们可以用state来声明变量，它会告诉Perl该变量属于当前子程序的私有变量，并且在多次调用这个子程序期间保留该变量的值。这个特性是从Perl 5.10开始引入的：

```
use 5.010;

sub marine {
    state $n = 0;  # 持久性私有变量$n
    $n += 1;
    print "Hello, sailor number $n!\n";
}
```

现在我们可以在用了strict编译指令并且不用全局变量的前提下得到和之前相同的输出。第一次调用该子程序时，Perl声明并初始化变量$n，而在接下来的调用中，这个表

达式将被Perl忽略。每次子程序返回后，Perl都会将变量$n的当前值保留下来，以备下次调用时再用。

类似标量变量，其他任意类型的变量都可以被声明为state变量。下面的子程序可以通过声明state数组来保留它的参数及计算总和：

```
use 5.010;

running_sum( 5, 6 );
running_sum( 1..3 );
running_sum( 4 );

sub running_sum {
  state $sum = 0;
  state @numbers;

  foreach my $number ( @_ ) {
    push @numbers, $number;
    $sum += $number;
  }

  say "The sum of (@numbers) is $sum";
  }
```

每次我们调用这个子程序的时候，它都会将新的参数与之前的参数相加，因此每次都会输出一个新的总和：

```
The sum of (5 6) is 11
The sum of (5 6 1 2 3) is 17
The sum of (5 6 1 2 3 4) is 21
```

但是在使用数组和哈希类型的state变量时，还是有一些轻微限制的。在Perl 5.10中我们不能在列表上下文中初始化这两种类型的state变量：

```
state @array = qw(a b c); # 错误！
```

这样做会报错，并提示将来版本的Perl也许支持这种方式。而现在到了Perl 5.14，我们还是无法这么用：

```
Initialization of state variables in list context currently forbidden ...
```

习题

以下习题答案参见第311页上的“第四章习题解答”一节：

1. [12]写一个名为total的子程序，它可以返回给定列表中数字相加的总和。提示：该子程序不需要执行任何I/O，它只需要按要求处理它的参数并给调用者返回一个

值就行了。用下面这个程序来检验一下你写完的子程序，第一次调用时返回的列表中数字的总和应该是25 。

```
my @fred = qw{ 1 3 5 7 9 };
my $fred_total = total(@fred);
print "The total of \@fred is $fred_total.\n";
print "Enter some numbers on separate lines: ";
my $user_total = total(<STDIN>);
print "The total of those numbers is $user_total.\n";
```

2. [5]使用之前程序中的子程序，计算从1加到1 000的总和。

3. [18]额外附加题：写一个名为&above_average的子程序，当给定一个包含多个数字的列表时，返回其中大于这些数平均值的数。（提示：另外写一个子程序，通过用这些数的总和除以列表中数字的个数来计算它们的平均值。）当你写完后，用下面的程序检验一下。

```
my @fred = above_average(1..10);
print "\@fred is @fred\n";
print "(Should be 6 7 8 9 10)\n";
my @barney = above_average(100, 1..10);
print "\@barney is @barney\n";
print "(Should be just 100)\n";
```

4. [10]写一个名为greet的子程序，当给定一个人名作为参数时，打印出欢迎他的信息，并告诉他前一个来宾的名字：

```
greet( "Fred" );
greet( "Barney" );
```

按照语句的顺序，它应该打印出:

```
Hi Fred! You are the first one here!
Hi Barney! Fred is also here!
```

5. [10]修改前面这个程序，告诉所有新来的人之前已经迎接了哪些人：

```
greet( "Fred" );
greet( "Barney" );
greet( "Wilma" );
greet( "Betty" );
```

按照语句的顺序，它应该打印出：

```
Hi Fred! You are the first one here!
Hi Barney! I've seen: Fred
Hi Wilma! I've seen: Fred Barney
Hi Betty! I've seen: Fred Barney Wilma
```

第五章

输入与输出

先前为了习题的需要我们已经介绍过输入/输出（input/output，简称I/O）的一些用法。现在则要涉及日常编程中会遇到的大多数（约80%）的I/O问题。对于标准输入、标准输出以及标准错误这几个概念，如果你早已熟知，那算是处于超前状态。要是不熟，读完本章也该可以迎头赶上。现在，就让我们先将“标准输入”当成“键盘”，将“标准输出”当成“屏幕”好了。

读取标准输入

读取标准输入流相当容易，我们在前面已经用过行输入操作符<STDIN>[注1]。在标量上下文中执行该操作时，将会返回标准输入中的下一行：

```
$line = <STDIN>;              # 读取下一行
chomp($line);                 # 截掉最后的换行符

chomp($line = <STDIN>);       # 习惯用法，效果同上
```

如果读到文件结尾（end-of-file），行输入操作符就会返回undef。这样的设计是为了配合循环使用，可以自然跳出循环：

```
while (defined($line = <STDIN>)) {
    print "I saw $line";
}
```

第一行程序代码做了许多事：读取标准输入，将它存入某个变量，检查变量的值是否被

注1：　我们称<STDIN>为行输入操作符，实际上它是针对文件句柄（*filehandle*）的行输入操作符（用尖括号表示）。关于文件句柄，我们将在本章稍后讨论。

定义，以及我们是否该执行while循环的主体（也就是还没遇到输入的结尾）。因此在循环主体内，我们会在$line变量里看到各行输入的内容[注2]。这是十分常见的操作，所以Perl顺理成章为它定义了一个简写，如下所示：

```
while (<STDIN>) {
    print "I saw $_";
}
```

Larry从一堆冷僻的符号中找到一对尖括号，创造出了这个简写。换句话说，从字面上来讲，它的意思是："读取一行标准输入，看它是不是为真（通常是）。若是真，就进入while循环，并在下次循环时忘记刚刚读入的那一行，进入下一行！"Larry为这个无用的语法赋予了有用的意义。

它实际上和前面的第一个循环相同：它告诉Perl将标准输入读进某个变量，（只要能读到内容，表示还没到达文件结尾）然后进入while循环。然而这次Perl并不是把它读入$line里，而是放进它的默认变量$_中，就好比执行了下面的代码：

```
while (defined($_ = <STDIN>)) {
    print "I saw $_";
}
```

继续看下去之前，我们必须讲清楚一件事：这个简写只有在最早的写法中才能正常运行。如果你将行输入操作符放在其他任何地方（特别是自成一行的语句），它并不会读取一行输入并自动存入默认变量$_。唯独while循环的条件表达式里只有行输入操作符的前提下，这个简写才起作用[注3]。假如条件表达式里放了其他东西，它就无法按你的预期运行了。

行输入操作符（<STDIN>）和Perl的默认变量（$_）之间并没有什么关联，只是在这个简写里，输入的内容会恰好存储在默认变量中而已。

然而，如果在列表上下文中调用行输入操作符，它会返回一个列表，其中包含（其余）所有的输入内容，每个列表元素代表一行输入内容：

```
foreach (<STDIN>) {
    print "I saw $_";
}
```

再次强调，行输入操作符和Perl的默认变量（$_）之间并没有任何关联，只是在这个例

注2：　可能你注意到了在这里我们没有使用chomp函数。一般我们把chomp写在循环体内的第一行，而不是放在循环的条件表达式中。在接下来的章节中，你会看到很多这样的用法。

注3：　好吧，其实某种程度上for循环就相当于while循环，所以这里的简写语法依然适用。

子里，foreach的默认控制变量是$_而已。在这个循环里，我们会在$_中看到各行输入的内容。

好像在哪里听过。没错，它和while循环的行为完全一样，不是吗？

两者的不同之处在于它们背后的运作方式。在while循环里，Perl会读取一行输入，把它存入某个变量并且执行循环的主体，接下来它会回头去寻找其他的输入行。但是在foreach循环里，行输入操作符会在列表上下文中执行（因为foreach需要逐项处理列表内容）。为此，在循环能够开始执行前，它必须先将输入全部读进来。假如输入来自400MB大小的Web服务器日志文件，它们的差异会十分明显！因此最好的做法通常是尽量使用while循环的简写，让它每次处理一行。

来自钻石操作符的输入

还有另外一种读取输入的方法，就是使用钻石[注4]操作符<>。它能让程序在处理调用参数（稍后我们会看到）时，提供类似于标准Unix工具程序的功能[注5]。如果你想用Perl编写类似*cat*、*sed*、*awk*、*sort*、*grep*、*lpr*等工具程序，钻石操作符将会是你的好帮手。但若你想让它处理更复杂的参数格式，钻石操作符可就帮不上忙了。

程序的调用参数（*invocation argument*）通常是命令行上跟在程序名后面的几个“单词”[注6]。在下面的例子里，命令行参数就是要依次处理的几个文件的名字：

```
$ ./my_program fred barney betty
```

这条命令的意思是：执行*my_program*命令（位于当前目录），然后它应该会处理文件*fred*，接着是文件*barney*，最后是文件*betty*。

若不提供任何调用参数，程序会从标准输入流采集数据。此话不够严格，因为有个例外：如果把连字符（-）当作参数，则表示要从标准输入读取数据[注7]。所以，假如

注4：　钻石操作符是Larry的女儿Heidi命名的，某天当Randal拿着他新写的Perl培训材料去Larry家给他看的时候，这个操作符还没有一个叫得出的名字。Larry也想不出来，8岁的Heidi灵机一动，说“它像钻石”，于是有了这个名字，谢谢Heidi！

注5：　不光是Unix系统，许多其他系统也借鉴了这种处理调用参数的方式。

注6：　当程序开始运行时，它会有一个有零个或多个参数的列表，这取决于运行它的程序。通常是由shell来启动这个程序，这时参数列表的组成就取决于你在命令行上的输入。稍后我们会看到可以用任意字符串来充当调用参数。由于它们一般出现在命令行上，所以我们称之为“命令行参数”。

注7：　可能你对Unix的一些东西还不熟悉：大多数标准的工具程序，像*cat*和*sed*，它们都把连字符（-）当作标准输入流。

调用参数是*fred-betty*，那么程序应该先处理文件*fred*，然后处理标准输入流中提供的数据，最后才是文件*betty*。

让程序以这种方式运行的好处，就是你可以在运行时指定程序的输入源。比方说，你不需要重写程序，就可以在管道（pipeline）里使用它（稍后会有更深入的讨论）。Larry为Perl加上这个功能，是想帮你轻松写出用起来类似标准Unix工具程序的程序，并且在非Unix系统上也能运行。事实上，Larry是为了让他自己的程序更符合标准Unix工具程序的习惯，才设计出这个功能的。因为每家厂商的工具程序在运行方式上不尽相同，所以Larry自己用Perl按照统一的用法重新实现了这些工具程序并安装到各台机器上。当然，前提是把Perl也移植到那些机器上。

钻石操作符是行输入操作符的特例。不过它并不是从键盘取得输入，而是从用户指定的位置读取[注8]：

```
while (defined($line = <>)) {
    chomp($line);
    print "It was $line that I saw!\n";
}
```

所以，假设这个程序运行时的调用参数是fred、barney和betty，输出结果就会是“It was [从文件*fred*取得的一行内容] that I saw!”、“It was [从文件*fred*取得的另一行内容] that I saw!”等，直到我们遇到文件*fred*的结尾为止。接下来，它会自动切换到文件*barney*，逐行输出它的内容，然后再换到文件*betty*。请注意，在切换到另一个文件时中间并没有间断，因为使用钻石操作符时就好像这些文件已经合并成一个很大的文件一样[注9]。钻石操作符只有在碰到所有输入的结尾时才会返回undef（然后我们就会跳出while循环）。

既然这只是行输入操作符的一种特例，因此我们可以使用先前看到的简写，将输入读取到默认的$_里：

```
while (<>) {
    chomp;
    print "It was $_ that I saw!\n";
}
```

这和前面的循环效果相同，但可以少打些字。你可能也注意到，我们使用了chomp的默认用法：不加参数时，chomp会直接作用在$_上。节约按键，从小地方做起！

注8：　输入可能来自键盘，也可能来自其他设备。

注9：　不管你是否在意，有一点需要说明，当前正在处理的文件名会被保存在特殊变量$ARGV中。如果当前是从标准输入获取数据，那么当前文件名就会是“-”（连字符）。

由于钻石操作符通常会处理所有的输入，所以一旦看到它出现在程序中的多处时，通常都是错误的。如果程序里同时出现两个钻石操作符，尤其是在使用第一个钻石操作符进行读取的while循环中接着使用第二个的话，可以肯定无法正常工作[注10]。在我们的经验里，当初学者在程序中放入第二个钻石操作符时，往往是想读取下一行输入，其实$_才是他们想要的东西。请记住，钻石操作符是用来读取输入的，而输入的内容可以在$_中找到（起码是在一般的默认情况下）。

假如钻石操作符无法打开某个文件并读入内容，便会显示相关的出错诊断信息，就像：

```
can't open wimla: No such file or directory
```

然后钻石操作符会自动跳到下一个文件，就像*cat*或其他标准工具程序的做法一样。

调用参数

从技术上来看，钻石操作符其实不会去检查命令行参数，它的参数其实不过是来自@ARGV数组。这个数组是由Perl解释器事先建立的特殊数组，其内容就是由命令行参数组成的列表。换句话说，它和别的数组没有不同（除了奇怪的全大写名称之外），只不过在程序开始运行时，@ARGV里就已经塞满了调用参数[注11]。

你可以像使用其他数组一样来使用@ARGV：你可以把元素从它里面shift出去，或用foreach逐项加以处理。你也可以检查是否有参数是以连字符（-）开头的，然后将它们当成调用选项处理（就像Perl对待它自己的-w选项一样）[注12]。

钻石操作符会查看数组@ARGV，然后决定该用哪些文件名，如果它找到的是空列表，就会改用标准输入流；否则，就会使用@ARGV里的文件列表。这表示程序开始运行之后，只要尚未使用钻石操作符，你就可以对@ARGV动点手脚。这样我们就可以处理三个特定的文件，不管用户在命令行参数中指定了什么：

```
@ARGV = qw# larry moe curly #;  # 强制让钻石操作符只读取这三个文件
while (<>) {
  chomp;
```

注10：如果你在使用第二个钻石操作符前重新初始化了特殊变量@ARGV，那就是可行的。我们将在下一小节学习变量@ARGV。

注11：C程序员可能会因此联想到argc（Perl中没有这个）以及该程序的名称在哪（它在Perl的特殊变量$0里，不包含在@ARGV中）。另外，调用程序的方式不同，这里的情况也可能不同。具体请参考*perlrun*文档。

注12：如果你需要的命令选项多于一两个，最好以标准的方式用相关的模块。请参阅Getopt::Long和Getopt::Std模块的文档，它们都是随Perl发行的标准模块。

```
    print "It was $_ that I saw in some stooge-like file!\n";
}
```

我们会在第十四章中介绍如何转换输入内容的编码时，看到有关@ARGV变量的更多例子 [注13]。

输出到标准输出

print操作符会读取后续列表中的所有元素，并把每一项（当然是一个字符串）依次送到标准输出。它在每一项之前、之后与之间都不会再加上额外的字符 [注14]。要是想在每个元素之间加上空白并在结尾加上换行符，你得这么做：

```
$name = "Larry Wall";
print "Hello there, $name, did you know that 3+4 is ", 3+4, "?\n";
```

当然，直接使用数组和使用数组内插在打印效果上是不同的：

```
print @array;     # 把数组的元素打印出来
print "@array";   # 打印出一个字符串（数组元素内插的结果）
```

第一个print语句会一个接一个地打印出数组中所有的元素，元素之间不会有空格。而第二个则不同，它只打印一个字符串，也就是@array在双引号中内插形成的字符串，也就是数组@array所有元素（以空格分隔）组成的字符串 [注15]。所以，如果@array的内容是qw/fred barney betty/ [注16]，那么第一个语句会输出fredbarneybetty，而第二个会输出以空格隔开的fred barney betty。

不过，在你打算总是使用第二种写法之前，请先想象@array包含了一串未截尾的输入行。也就是说，请想象里面的每个字符串都是以换行符结尾。这时，第一个print语句会分三行输出fred、barney和betty。但是第二个print语句则会输出这样的结果：

```
fred
 barney
 betty
```

注13：有关字符编码的内容，如果你还不太熟悉，请先读一读附录C。

注14：默认情况下它不会额外添加什么东西，但这种默认情况（和Perl里很多其他默认情况一样）是可以被修改的。对默认情况的修改会增加维护人员的困难，所以除非是在短小精悍、需要快速完成功能的程序中，或者是在某一小段代码里，否则一般请不要修改这些默认情况。需要详细了解，请参看*perlvar*文档。

注15：空格也是默认情况而已，请再次参看*perlvar*文档。

注16：这是Perl的一种写法，它代表包含三个元素的列表，你知道的，对吧？

你看得出其中的空格是从哪里来的吗？因为Perl把数组内插到字符串中时，会在每个元素之间加上空格。因此我们得到的字符串将会是数组的第一个元素（fred与换行符），接着是一个空格，之后是数组的下一个元素（barney与换行符），接着是一个空格，之后是数组的最后一个元素（betty与换行符）。结果就是，除了第一行，其他几行看起来好像经过缩排一样。

每隔一两周，在新闻组或论坛中就会出现类似这样的主题的新帖子："Perl会在第一行之后缩进剩下的行"。

我们甚至不必看正文，马上就知道他的程序用了双引号里的数组，而数组里全是没有截尾过的字符串。

我们会问"你是不是在双引号里放了数组，而数组里面是没chomp过的字符串？"而答案常常都是肯定的。

一般来说，如果数组里的字符串包含换行符，那么只要直接将它们输出来就好了：

```
print @array;
```

要是它们不包含换行符，你通常会想在结尾补上一个：

```
print "@array\n";
```

为了帮你分清楚哪个是哪个，通常在使用引号的场合，字符串后面都最好加上\n。

一般情况下，程序的输出结果会先被送到缓冲区。也就是说，不会每当有一点点输出就直接送出去，而是先积攒起来，直到数量够多时才造访外部设备。

为什么要这样做呢？举例来说，当输出结果要存到磁盘时，只为了添加一两个字符到文件结尾就去访问磁盘，（相对来讲）这样既慢又没效率。因此，输出的结果通常会先被送到缓冲区，等到缓冲区满了或是在输出结束时（例如程序运行完毕），才会将它刷新(*flush*)到磁盘（也就是实际写到磁盘，或者写到其他地方）。通常这就是你想要的效果。

但假如你（或程序）立刻就要输出，你大概愿意牺牲一些效率，在每次print时立刻刷新缓冲区。在这种情况下，请参阅Perl的文档，进一步了解如何控制缓冲。

由于print处理的是待打印的字符串列表，因此它的参数会在列表上下文中被执行。而钻石操作符（行输入操作符的特殊形式）在列表上下文中会返回由许多输入行组成的列表，所以它们彼此可以配合工作：

```
print <>;           # 相当于Unix下的cat命令
print sort <>;      # 相当于Unix下的sort命令
```

老实说，*cat*和*sort*这两个标准的Unix命令还有许多额外的功能是上面两行程序代码做不到的。但它们绝对是物超所值！现在你可以用Perl重写所有的Unix工具程序，然后轻松地将它们移植到任何有Perl的机器上，不管那台机器上的操作系统是否为Unix。在各种不同的机器上，你都可以保证程序会用相同的方式运行[注17]。

有个比较不明显的问题，那就是`print`后面可有可无的括号，这可能会让人糊涂。别忘了有这条规则：除非这样做会改变表达式的意义，否则Perl里的括号可以省略。所以有两种方法可以输出一样的东西：

```
print("Hello, world!\n");
print "Hello, world!\n";
```

到目前为止还不错。不过Perl还有另一条规则：假如`print`的调用看起来像函数调用，它就是一个函数调用。这个规则很简单，但是“看起来像函数调用”是什么意思呢？

在函数调用里，函数名后面必须紧接着[注18]一对括号，里面包含了函数的参数，就像这样：

```
print (2+3);
```

这看起来像函数调用，所以它的确是个函数调用。它会输出5，但它和其他函数一样返回某个值。`print`的返回值不是真就是假，代表`print`是否成功执行。除非发生了I/O错误，否则它一定会成功。所以在下面的语句里，`$result`通常会是1：

```
$result = print("hello world!\n");
```

可是，如果你用别的方式来处理这个结果，会发生什么事呢？假设你决定将返回值乘以4：

```
print (2+3)*4;  # 糟糕！
```

当Perl看到这行程序代码，它会遵照你的要求输出5。接着，Perl会从`print`取得返回值

注17：实际上Perl Power Tools（简称 PPT）项目的目标就是用Perl来重写所有经典Unix工具程序（甚至包括游戏！），但是在重写shell的时候陷入了难题。PPT项目一度非常有用，因为它使得所有标准的工具程序能运行在很多非Unix机器上。

注18：这里我们说“紧接着”是因为在调用函数的时候，Perl不允许函数名和括号之间有换行符。如果有换行符，Perl 会把括号当成创建列表的操作符，而不是函数调用的一部分。我们只是想让你知道这些琐碎的细节，如果你实在好奇，可以在说明文档里面寻找更详细的内容。

1，再将它乘以4。然后它会丢掉这项乘积，因为你没告诉它接下来要做什么。这时，你旁边就会有人看到，然后说："嘿，Perl连数学都不会！应该输出20，而不是5！"

问题在于Perl可以省略括号，而大多数人又容易忘记括号的归属。没有括号的时候，print是列表操作符，会把其后的列表里的所有东西全都输出来。一般来说，这就是我们想要的。但是假如print后面紧跟着左括号，它就是一个函数调用，只会将括号内的东西输出来。因为该行程序代码有括号，所以对Perl来说，就和下面的写法一样：

```
( print(2+3) ) * 4;  # 糟糕!
```

好在只要你启用了警告功能，Perl几乎总能帮你找出这类问题。所以请使用-w或加上use warnings，至少在你开发程序与调试时启用它。

实际上，这条规则——"假如它看起来像函数调用，它就是一个函数调用"，不仅对print适用，对Perl所有的列表函数也同样适用[注19]，只不过print可能最容易让人注意到这条规则。如果print（或其他函数名）后面接着一个左括号，请务必确定在函数的所有参数之后也有相应的右括号。

用printf格式化输出

处理输出结果时，你也许希望使用控制能力比print更强的操作符。事实上，你可能已经习惯使用C的printf函数来产生格式化过的输出结果。别担心，Perl里同名的函数也能提供类似的功能。

printf操作符的参数包括"格式字符串"及"要输出的数据列表"。格式[注20]字符串好像用来填空的模板，代表你想要的输出格式：

```
printf "Hello, %s; your password expires in %d days!\n",
    $user, $days_to_die;
```

格式字符串里可以有多个所谓的转换（*conversion*）。每种转换都会以百分比符号（%）开头，然后以某个字母结尾（我们稍后就会看到，在这两个符号之间可以存在额外的有效字符）。而后面的列表里元素的个数应该和转换的数目一样多，如果数目不对，就无法正确运行。上面的例子里有两个元素和两个转换格式，所以输出的结果看起来会像这样：

注19：　不需要参数或者只需要一个参数的函数不包括在内。

注20：　我们这里说的"格式"只是一般意义上的。Perl本身还有个报表格式化（format）功能，但我们现在不会介绍它的用法，你可以到附录B参考一下。我们希望由简入难，以免两者混淆。

```
Hello, merlyn; your password expires in 3 days!
```

printf可用的转换格式很多，所以我们在这里只会说明最常用的部分。当然，你可以在*perlfunc*文档里找到详细的说明。

要输出恰当的数字形式，可以使用%g [注21] ，它会按需要自动选择浮点数、整数甚至是指数形式：

```
printf "%g %g %g\n", 5/2, 51/17, 51 ** 17;  # 2.5 3 1.0683e+29
```

%d格式则代表十进制 [注22] 整数，它会舍去小数点之后的数字：

```
printf "in %d days!\n", 17.85;  # 输出：in 17 days!
```

请注意，它会无条件截断，而非四舍五入。等一下我们就会学到如何四舍五入。

在Perl里，printf最常用在字段式的数据输出上，因为大多数的转换格式都可以让你指定宽度。如果数据太长，字段会按需要自动扩展：

```
printf "%6d\n", 42;  # 输出结果看起来像````42 （` 符号代表空格）
printf "%2d\n", 2e3 + 1.95;  # 2001
```

%s代表字符串格式，所以它的功能其实就是字符串内插，只是还能设定字段宽度：

```
printf "%10s\n", "wilma";  # 看起来像`````wilma
```

如果宽度字段是负数，则会向左对齐（适用于上述各种转换）：

```
printf "%-15s\n", "flintstone";  # 看起来像flintstone`````
```

%f转换格式（浮点数）会按需要四舍五入，甚至还可以指定小点数之后的输出位数：

```
printf "%12f\n", 6 * 7 + 2/3;    # 看起来像```42.666667
printf "%12.3f\n", 6 * 7 + 2/3;  # 看起来像``````42.667
printf "%12.0f\n", 6 * 7 + 2/3;  # 看起来像``````````43
```

要输出真正的百分号，请使用%%，它的特殊之处在于不会输出（参数）列表中的任何元素 [注23] ：

注21：你可以把“g”当成“General”数字转换，或“Good conversion for this number”，或“Guess what I want the output to look like”。

注22：如果你需要，还有%x代表十六进制，%o代表八进制。你可以把%d当作“decimal”，以帮助记忆。

注23：也许你认为在百分号前加个反斜线就行了。不错的尝试，但是不对。这是因为这些格式符号是表达式，而表达式"\%"就代表字符串'%'。所以就算我们加了反斜线，printf还是不知道该怎么做。一般 C 程序员比较习惯这么做。

```
printf "Monthly interest rate: %.2f%%\n",
    5.25/12;  # 运算后的值看起来像 "0.44%"
```

数组和printf

一般来说，你不会将数组当成printf的参数。这是因为数组可以包含任意数目的元素，而格式字符串却只会用到固定数目的元素：假如格式字符串里有三个转换格式，那就必须刚好有三个元素可供使用。

不过，没有人规定你不能在程序运行时动态产生格式字符串，它们可以是任意的表达式。但是要想做得正确需要一些技巧，把格式字符串存进变量可能会带来方便（尤其有助于调试）：

```
my @items = qw( wilma dino pebbles );
my $format = "The items are:\n" . ("%10s\n" x @items);
## 打印 "the format is >>$format<<\n"; # 用于调试
printf $format, @items;
```

这段程序用了x操作符（参见第二章）复制指定的字符串，复制的次数与@items（在标量上下文中使用）的元素个数相同。在上面的例子里，因为列表有三个元素，所以复制次数是3。这样一来，产生的格式字符串就和直接写“The items are:\n%10s\n%10s\n%10s\n”一样。程序会先输出标题，接着将每个元素显示成独立的一行，每行都靠右对齐，字段一律10个字符宽。很酷吧？这还不算，因为最酷的是我们可以把它们全都组合在一起：

```
printf "The items are:\n".("%10s\n" x @items), @items;
```

请注意，我们在标量上下文中用了一次@items以取得它的长度，然后又在列表上下文中用了一次以取得它的内容。上下文的重要性可见一斑。

文件句柄

文件句柄（filehandle）就是程序里代表Perl进程（process）与外界之间的I/O联系的名称。也就是说，它是“这种联系”的名称，不是文件的名称。

在Perl 5.6之前，所有的文件句柄名称都是裸字（bareword），而从Perl5.6起，我们可以把文件句柄的引用放到常规的标量变量中。我们会先展示裸字写法，因为许多特殊文件句柄向来习惯使用裸字，稍后我们再介绍存放在标量变量中的文件句柄的用法。

给文件句柄起名就好比给Perl的其他标识符起名一样，必须以字母、数字及下划线组成，但不能以数字开头。由于裸字没有任何前置字符，所以当我们阅读它的时候，有

可能会与现在或将来的保留字相混淆，或是与第十章中将要介绍的标签（label）相混淆。所以再一次，Larry建议你使用全大写字母来命名文件句柄。这样不仅看起来更加明显，也能避免与将要引入的（小写）保留字冲突，以免程序出错。

但有6个特殊文件句柄名是Perl保留的，它们是：STDIN、STDOUT、STDERR、DATA、ARGV以及ARGVOUT [注24]。虽然你可以选择任何喜欢的文件句柄名，但不应使用保留字，除非确实需要以特殊方式使用上述6个句柄 [注25]。

上述的文件句柄也许你早就认得了。当程序启动时，文件句柄STDIN就是Perl进程和它的输入源之间的联系，也就是俗称的标准输入流(*standard input stream*)。它通常是用户的键盘输入，除非用户要求别的输入来源，像从文件读取输入或是经由管道（pipe）读取另一个程序的输出 [注26]。当然还有标准输出流(*standard output stream*)STDOUT。默认情况下它会输出到用户的屏幕，但用户也可以把它送到文件或另一个程序，我们稍后会看到这种范例。这些标准的流乃是源自于Unix的“标准I/O”函数库，不过大部分现代的操作系统也都支持 [注27]。一般来说，程序应该不闻不顾地从STDIN读取数据，继而不闻不顾地将数据写到STDOUT，相信用户（或者广义地说，是启动你的程序的那个程序）已经将它们都设定好了。比如说，用户可以在shell里运行以下命令：

```
$ ./your_program <dino >wilma
```

这条命令告诉shell，程序的输入应该来自文件*dino*，输出应该送到文件*wilma*。而这个程序本身只管从STDIN读入数据，再按照我们的要求处理这些数据，之后只要把输出送到STDOUT即可，别的什么都不用管，交给shell来处理。

这样不需要额外的工作，这个程序就可以正确地在管道（pipeline）中运行。这又是另一个来自Unix的概念，它可以让我们将命令写成串联的形式：

```
$ cat fred barney | sort | ./your_program | grep something | lpr
```

注24：　也许有些人讨厌用大写字母，哪怕一会儿，所以他们可能喜欢用小写字母来写这些名字，比如stdin。Perl也许能让你这么做，但是不保证你总这么做是不会出错的。至于何时可以，何时不行，已经超出了本书范围。但重要的是，如果你老用小写，说不定什么时候就出错了，所以这些文件句柄名还是别用小写吧。

注25：　在某些情况下，你可以重用这些名称。但是维护你的程序的人可能会错误地认为你在使用这些内置的保留文件句柄名，他会很容易混淆的。

注26：　我们这里所说的那三个默认 I/O 流是针对Unix shell，而不是指启动某个程序的shell。我们会在第十四章中了解用Perl启动其他程序的时候会发生什么。

注27：　如果你对非Unix系统的标准输入和输出不熟悉，请参阅*perlport*文档或者你的系统说明文档中讲解 Unix shell（或类似的程序，它们能根据你键入的命令调用其他程序）的章节。

假如你不熟悉这些Unix命令，那也没有关系。上面这条命令的意思是由*cat*命令将文件*fred*的每一行输出，再加上文件*barney*里的每一行。之后，将以上输出作为*sort*命令的输入，对所有的行进行排序，继而把排序结果交给*your_program*处理。*your_program*完成相应操作后，再将输出数据送到*grep*，由它过滤掉数据中的某些行，并将剩下的数据输出到*lpr*这个命令，让它负责把最终结果传送给打印机打印出来。一气呵成，是不是很酷？

在Unix以及许多其他现代的操作系统里，上述的管道用法相当常见。这样一来，你只要用好几个简单、标准的组件，就能构造出功能强大且复杂的命令。每个组件都能把一件事情做得很好，至于如何巧妙地组合起来就是你的任务了。

除了标准输入和标准输出，还有另一个标准I/O流。如果（在上一个例子中）*your_program*发出任何警告或是其他诊断信息，这些信息就不应该继续在管道中往下传递。我们已经用*grep*命令来筛除特定字符串以外的任何数据，因此它很可能会丢弃警告信息。而且即使它留下了警告信息，我们可能也不想让它传递到管道中下游的其他程序。这就是为什么我们还有一个标准错误流(*standard error stream*)：STDERR。即使输出会被重定向到下一个程序或文件，错误信息仍能流向用户需要之处。错误信息在默认情况下通常是输出到用户的屏幕[注28]，但用户还是可以用下面这样的shell命令将错误信息转向某个文件：

```
$ netstat | ./your_program 2>/tmp/my_errors
```

打开文件句柄

到目前为止，你已经看到过三种Perl提供的默认文件句柄——STDIN、STDOUT以及STDERR——它们都是由产生Perl进程的父进程（可能是shell）自动打开的文件或设备。当你需要其他文件句柄时，请使用open操作符告诉Perl，要求操作系统为你的程序和外界架起一道桥梁。来看几个具体例子：

```
open CONFIG, 'dino';
open CONFIG, '<dino';
open BEDROCK, '>fred';
open LOG, '>>logfile';
```

第一行会打开名为CONFIG的文件句柄，让它指向文件*dino*。换句话说，它会打开（已存在的）文件*dino*，文件中的任何内容都能从文件句柄CONFIG被读到我们的程序中来。这

注28：　另外，错误信息在一般情况下是不会被缓冲的。标准错误流和标准输出流会指向同一个地方（比如显示器），而错误信息可能在正常的输出之前。举例来说，如果你的程序打印一行常规的文本，然后试图除以零，这个时候与除以零相关的错误信息就会比常规文本先输出到屏幕。

和利用shell的重定向（例如<dino这样的写法）将文件内容经STDIN读入程序的做法相似。事实上，第二行正好用到了这样的技巧，它和第一行所做的事完全相同，只不过用了小于号来声明“此文件只是用来读取的，而非写入”。打开文件句柄的默认模式就是读取数据[注29]。

尽管你无须使用小于号来打开一个文件表示读取输入，但我们还是提到了它，因为第三个例子有个与之相对的大于号，它可以用来创建一个新的文件：它会打开文件句柄BEDROCK并输出到新文件*fred*。大于号的用途跟shell的重定向一样，它会将输出送到一个名为*fred*的新文件。如果已经存在一个名为*fred*的文件，那么就清除原有的内容并以新内容取代之。

第四个例子展示了如何使用两个大于号指明以追加的方式来打开文件（这仍然和shell的重定向方式相同）。换句话说，如果文件原本就存在，那么新的数据将会添加在原有文件内容的后面；如果它不存在，就会创建一个新文件，和只有一个大于符号的情形相同。此特性对于日志文件（logfile）来说非常方便，你的程序可以在每次运行时只加几行数据到日志文件里。这就是为什么第四个例子的文件句柄称为LOG，而文件名是*logfile*。

你可以使用任意一个标量表达式来代替文件名说明符。不过，你通常会想要明确指定输入或输出的方向：

```
my $selected_output = 'my_output';
open LOG, "> $selected_output";
```

注意大于号后的空格。Perl会忽略它[注30]，但这个空格能防止意外发生。如果$selected_output的值是">passwd"而之前又没有空格的话，就会变成以替换方式写入，而非以追加方式写入文件。

在相对新版的Perl（5.6版之后）里，open还有一种三个参数形式的写法：

```
open CONFIG, '<', 'dino';
open BEDROCK, '>', $file_name;
```

注29：选择读取操作作为默认行为是出于安全的考虑。随后我们会在第十四章详细介绍文件名中可能用到的其他几个魔法字符。如果$name里是由用户选择的文件名，那么直接使用变量$name打开文件，就有可能把文件名中出现的魔法字符当作特定功能，执行意料之外的操作。所以我们建议你，不管什么时候，都要以三个参数的形式打开文件句柄。我们稍后就会具体介绍这种形式的写法。

注30：是的，这意味着如果你的文件名以空格开头，Perl同样会忽略掉这些空格。如果你担心这方面的问题，请参阅*perlfunc*和*perlopentut*文档。

```
open LOG, '>>', &logfile_name();
```

其优点在于语法上可以很容易区分模式（第二个参数）与文件名本身（第三个参数），这种写法在安全方面有些好处。

其实除此之外，三个参数形式的写法还有一个好处，那就是有机会指定数据的编码方式。如果你预先知道要读取的文件是UTF-8编码的，则可以在文件操作模式后面加上冒号，然后写上编码名称：

```
open CONFIG, '<:encoding(UTF-8)', 'dino';
```

反过来，如果要以特定编码写数据到某个文件，也可以这么用：

```
open BEDROCK, '>:encoding(UTF-8)', $file_name;
open LOG, '>>:encoding(UTF-8)', &logfile_name();
```

对此我们还有一个简便写法，不用每次键入`encoding(UTF-8)`，只要写上`:utf8`就可以了。但两者之间并不完全等同，简写方式不会考虑输入或输出的数据是否真的就是合法的UTF-8字符串。如果用`encoding(UTF-8)`的话，它是会确认编码是否正确这一点的。`:utf8`则不管拿来的是什么，直接当作UTF-8编码字符串来处理，所以有时候会出现一些问题。即便如此，还是有很多人喜欢这么用：

```
open BEDROCK, '>:utf8', $file_name;  # 可能会有问题
```

使用`encoding()`的形式，还能指定其他类型的编码。我们可以通过下面这条单行命令打印出所有*perl*能理解和处理的字符编码清单：

```
% perl -MEncode -le "print for Encode->encodings(':all')"
```

这个列表中出现的名字应该都可以拿来用在读取和写入文件时指定编码。但有些编码方式可能在其他机器上无法使用，关键还是要看相应的编码系统是否安装在系统中。

如果想要little-endian版本的UTF-16字符串，可以这样写：

```
open BEDROCK, '>:encoding(UTF-16LE)', $file_name;
```

或者是Latin-1字符集：

```
open BEDROCK, '>:encoding(iso-8859-1)', $file_name;
```

除了转换字符编码之外，数据输入或输出过程当中还有其他层（*layer*）[注31]可以控制

注31：　层的概念和编码转换略有不同，它实际上并不做什么事情。我们可以选择不同的层叠加起来（就是因为能叠加才叫做层），产生不同的效果。

数据的转换操作。比如说，有时候你拿到的文件采用DOS风格的换行符，也就是文件中每行都以回车换行（carriage-return/linefeed，简写为CR-LF）对（也常写作"\r\n"）结尾。而Unix风格的换行符则只是一个"\n"。不管把谁当作谁，弄错了的话出来的结果就会很怪异。借助:crlf层，我们就可以自动解决这个问题[注32]。如果想要确保得到的文件每行都以CR-LF结尾，就得在操作该文件时使用这个特殊层：

```
open BEDROCK, '>:crlf', $file_name;
```

现在每行写入文件时，该层就会把每个换行符转换为CR-LF。不过请注意，如果原本就是CR-LF风格的话，转换后会多出一个换行符。

读取DOS风格的文件时也可以这样转换：

```
open BEDROCK, '<:crlf', $file_name;
```

读入文件的同时，Perl会把所有CR-LF都转换为Unix风格的换行符。

以二进制方式读写文件句柄

在处理数据之前，其实不必预先知道它的实际编码方式，就算知道也不必每次都写明。在以前比较旧的Perl版本中，如果不希望转换换行符，比如某个二进制文件中恰好有一段字节顺序和换行符的内码相同，可以用binmode关闭换行符相关的处理[注33]：

```
binmode STDOUT; # 不要转换行符
binmode STDERR; # 不要转换行符
```

从Perl5.6开始，你可以在binmode的第二个参数的位置上指定层[注34]。如果你希望输出Unicode到STDOUT，就要确保STDOUT知道如何处理它拿到的数据：

```
binmode STDOUT, ':encoding(UTF-8)';
```

如果不这么写的话，会得到警告信息（就算没有启用警告功能也会），因为STDOUT不知道该如何处理编码上的问题：

```
Wide character in print at test line 1.
```

注32：幸运的是，:crlf编码在Windows系统上也是默认就支持的。

注33：这和在 FTP 客户端中设置二进制传输模式的概念极为相似，如果你还记得这个东西的话。

注34：Perl 5.6 把它称为变换规则（*discipline*），但之后改称为层（*layer*）。

实际上，不管输入还是输出，都可以用binmode指定特定行为。如果传到标准输入的是UTF-8编码的字符，那么应该事先告诉Perl按照UTF-8的方式处理：

```
binmode STDIN, ':encoding(UTF-8)';
```

有问题的文件句柄

Perl和其他程序语言一样，它自身是无法打开系统中的文件的，只能要求操作系统代劳。当然，操作系统也可能会以权限不足、文件名错误等理由拒绝打开。

如果试着从有问题的文件句柄（即没有正确打开的文件句柄或关闭的网络连接）读取数据，会立刻读到文件结尾（对于你将会在本章看到的各种I/O方法而言，“文件结尾”在标量上下文中是undef，在列表上下文中则是空列表）。如果试图将数据写进有问题的文件句柄，这些数据将会被无声无息地丢弃。

幸好，这种可怕的情况能轻易避免。首先，如果我们一开始就用-w选项或者warnings编译指令来启用警告功能的话，那么在用到有问题的文件句柄时，Perl会发出警告。但即使没有启用警告功能，open的返回值也能告诉我们它的执行结果成功与否：返回真表示成功，返回假则表示失败。所以，程序可以写成这样：

```
my $success = open LOG, '>>', 'logfile';  # 捕获返回值
if ( ! $success ) {
    # open 操作失败
    ...
}
```

当然，你可以照搬这种写法，不过稍后我们还会看到更简单流畅的写法。

关闭文件句柄

当你不再需要某个文件句柄时，可以用close操作符来关闭它：

```
close BEDROCK;
```

所谓关闭文件句柄，就是让Perl通知操作系统，我们对该数据流的处理已经全部完成，所以请系统将尚未写入的输出数据写到磁盘，以免有人等着使用[注35]。当你重新打开

注35：假如你熟悉I/O系统，你应该知道这里还有很多细节。一般来说，当文件句柄关闭的时候情况会是这样的：如果文件中仍有输入，它会被忽略；如果管道中仍有输入，那么对这个管道进行写操作的程序会收到管道被关闭的信号；如果有到文件或者管道的输出，那么缓冲区会被刷新（也就是保存在缓冲区的内容会被马上发送出去）；如果文件句柄加了锁，那么锁会被释放。更多的细节，请参阅相关操作系统的文档，特别是关于 I/O 的章节。

某个文件句柄时（也就是说，在新的open命令中重用之前的文件句柄名），Perl会自动关闭原先的文件句柄。在程序结束时，Perl也会自动关闭文件句柄[注36]。

正因为Perl这么贴心，所以很多简单的小程序都不必操心关闭的问题。但是若你想要写得工整些，请为每个open搭配一个close。最好是在每个文件句柄用完之后就立刻关闭它，哪怕程序马上就结束了[注37]。

用die处理致命错误

让我们先稍稍岔开一下话题。目前需要一种不直接相关于（或者说不只用于）I/O的机制，用于在异常发生时提前退出程序。

当Perl遇到致命错误时（例如：除以零，使用不合法的正则表达式，或调用未定义的子程序等等），你的程序应该立刻中止运行，并发出错误信息告知原因[注38]。这样的功能可以用die函数来实现，它让我们能够自己触发致命错误并给出错误消息。

die函数会输出你指定的信息到专为这类信息准备的标准错误流中，并且让你的程序立刻终止并返回不为零的退出码。

你可能还不知道，每个在Unix（以及许多其他现代操作系统）上运行的程序都会有一个退出状态（exit status），用来通知操作系统该程序的运行是否成功。那些以调用其他程序为工作内容的程序（比如工具程序*make*）会查看那些程序的结束状态来判断是否一切顺利。所谓的“结束状态”其实只用一个字节来表示，所以它能传递的信息不多。传统上，零代表成功，非零代表失败。也许“1”代表命令参数中的语法错误，“2”代表处理某程序时发生了错误，而“3”则可能代表找不到某个配置文件，各个程序的细节不尽相同。但是，“0”一定代表程序顺利完成。像*make*这样的程序，在看到失败的结束状态时就不会再执行下一步了。

注36：退出程序会关闭所有的文件句柄，但是如果Perl本身出错了，那么缓冲区的内容不会被刷新。也就是说，如果你的程序由于除以零而意外崩溃了，Perl仍会在运行，它仍能保证你的数据及时地得到输出。但是如果Perl本身无法运行了（比如内存不足或者收到了意外信号），那么最后的一点数据可能就无法及时得到处理（输出或写到磁盘）。不过，这通常不是什么大问题。

注37：关闭文件句柄会刷新输出缓冲并释放该文件上的锁。因为可能有人等着用这些文件，所以需要长时间运行的程序最好尽可能快地关闭它打开的文件句柄。但是我们很多的程序也许只需要一两秒就运行完了，所以这种情况下你不关闭也没问题。关闭文件句柄同样也会释放可能的有限资源，所以它不仅仅只是为了程序看起来工整而已。

注38：当然，这是默认情况。但错误也可能被eval语句块捕获，稍后我们会在第十七章中作进一步介绍。

所以，我们可以将前面的例子改写成这样：

```
if ( ! open LOG, '>>', 'logfile' ) {
    die "Cannot create logfile: $!";
}
```

如果open失败，die会终止程序的运行，并且告诉我们无法创建日志文件。可是冒号后面的$!代表什么呢？那就是可读的系统错误信息。一般来说，当系统拒绝我们请求的服务时（比如打开某个文件），$!会给我们一个解释。在这个例子里，也许是“权限不足（permission denied）”或“文件找不到（file not found）”之类的，也就是你在C或是其他类似的语言里调用perror取得的字符串。这个解释性的系统错误信息就保存在Perl的特殊变量$!中[注39]。因此，将$!放到错误信息中帮助用户了解自己遇到了什么问题，这是个不错的主意。不过，倘若你用die函数来显示程序中不属于系统服务请求的错误，这种时候请不要在信息里加上$!，因为这时它保存的可能只是Perl底层操作导致的无关信息。只有在系统服务请求失败后的瞬间，$!的值才会有用。如果操作成功了，就不会在$!里留下任何有用的信息。

die还会帮你做一件事。它会自动将Perl程序名和行号[注40]附加在错误信息的后面，因此你就可以轻易判断出程序里的哪个die函数才是造成程序过早结束运行的原因。在上一个例子里，可能会看到如下错误信息（假设$!的内容是permission denied）：

```
Cannot create logfile: permission denied at your_program line 1234.
```

这非常有用！事实上，每当我们想要了解错误信息的内幕，总会需要程序代码行号。如果你不想显示行号和文件名，请在die函数中的错误信息尾端的加上换行符。也就是说，die的另一种用法就是加上结尾的换行符，形式如下所示：

```
if (@ARGV < 2) {
    die "Not enough arguments\n";
}
```

如果命令行参数不足两个，范例程序会显示这行信息并中止运行。因为行号在此处对用户并没有用处（毕竟这是用户导致的错误），所以这里并不会显示程序名和行号。一个

注39：在某些非Unix的操作系统中，$!可能包含类似error number 7的信息，从而让用户去相关的文档中查看到底是哪里出错了。在Windows和VMS系统中，特殊变量$^E可能会包含一些附加的诊断信息。

注40：如果在读取文件块的时候发生了错误，那么出错信息还会包含“块编号”（一般是行号）和文件句柄名，这些信息对于跟踪bug很有用。

建议就是，用来指示用法错误的信息里可以加上结尾的换行符，但是若想在调试过程中追踪相关的错误，就不要加上结尾的换行符[注41]。

请一定记得检查open的返回值，因为之后的程序代码必须在文件打开成功时才能顺利运行。

用warn送出警告信息

恰如die函数的功能是产生像Perl的内置错误这样的致命错误（比如除以零）一样，warn函数的功能就是产生类似于Perl的内置警告信息的信息（比如启用警告信息时，使用某个undef变量却将它当成已有值来参与运算，就会触发警告信息）。

warn函数的功能就和die函数差不多，不同之处在于最后一步，它不会终止程序的运行。如有需要，它也可以加入程序名与行号，而且它会将信息送到标准错误流，这点是和die函数一致的[注42]。

在讨论过如何发出致命错误信息和普通警告信息之后，现在回到有关I/O的事情上来，请继续阅读。

自动检测致命错误

从Perl 5.10开始，为人称道的autodie编译指令已经成为标准库的一部分。像下面这个例子，原来的写法是自己检查open的返回值并处理错误：

```
if ( ! open LOG, '>>', 'logfile' ) {
    die "Cannot create logfile: $!";
}
```

每次打开一个文件句柄都要这么写一遍的话，无疑是十分繁琐的。现在有了autodie编译指令，这部分工作便得以解放。如果open失败，它会自动启动die：

```
use autodie;
```

注41： 程序名保存在Perl的特殊变量$0中，你可以把它加在这行中：$0:Not enough arguments\n，当你的程序在管道或者shell脚本中被使用，却不清楚是其中哪个命令正在抱怨的时候，这样做会非常有用。但是，在程序执行的时候$0还是可以被改变的。另外你可能还想了解一下__FILE__和__LINE__这两个特殊记号（或者caller函数），这样你就可以自己写一行语句来输出符合你的格式要求的信息。

注42： 和致命错误不同，警告是无法被eval捕获的。如果确实想要捕获警告的话，请参阅文档中有关__WARN__的章节（在*perlvar*文档里面谈到%SIG的部分）。

```
open LOG, '>>', 'logfile';
```

这条编译指令是靠判别具体操作的类型来工作的。如果Perl内置函数的幕后操作需要调用操作系统接口的话，那么中途出现的错误并不是编程人员所能控制的，所以一旦发现这部分系统调用出错，autodie便会自动帮你调用die，而它发出的错误信息也大体和我们自己组织的不相上下：

```
Can't open('>>', 'logfile'): No such file or directory at test line 3
```

使用文件句柄

一旦文件句柄以读取模式打开后，便可以从它读取一行行数据，就像从STDIN读取标准输入流中的数据一样。来看读取Unix系统密码文件的例子：

```
if ( ! open PASSWD, "/etc/passwd") {
    die "How did you get logged in? ($!)";
}

while (<PASSWD>) {
    chomp;
    ...
}
```

在这个例子里，die的信息中用了一对括号围住$!。它们只不过是括住输出信息的括号而已（有时候标点符号就只是标点符号）。正如你看到的，所谓的“行输入操作符”是由两部分组成的：一对尖括号（真正的行输入操作符）以及里面用来输入的文件句柄。

以写入或添加模式打开的文件句柄可以在print或printf函数中使用。使用时，请直接将它放在函数名之后、参数列表之前：

```
print LOG "Captain's log, stardate 3.14159\n";  # 输出到文件句柄 LOG
printf STDERR "%d percent complete.\n", $done/$total * 100;
```

你注意到文件句柄和要输出的内容之间没有逗号了吗[注43]？有括号时，它看起来会更奇怪。但下面两种写法都没错：

```
printf (STDERR "%d percent complete.\n", $done/$total * 100);
printf STDERR ("%d percent complete.\n", $done/$total * 100);
```

注43：　如果你对英语或者语言学很在行，当我们说这种情况叫做“间接对象语法”的时候，你可能会说：“哦！当然拉！句柄名后面没有跟逗号——所以它是间接对象！”但是其实我们并不懂为什么这里就不用逗号，之所以省略逗号是因为Larry说这个逗号是可以省略的。

改变默认的文件输出句柄

默认情况下，假如你不为print（或是printf，我们下面的说明对两者都有效）指定文件句柄，它的输出就会送到STDOUT。不过，你可以使用select操作符来改变默认的文件句柄。请看下面的例子，我们把默认输出改成BEDROCK这个文件句柄：

```
select BEDROCK;
print "I hope Mr. Slate doesn't find out about this.\n";
print "Wilma!\n";
```

一旦选择(select)了输出用的默认文件句柄，程序就会一直往那里输出。但是，这么做很容易使余下的程序变得混淆，所以这并不是一个好办法。因此，当你所指定的默认文件句柄使用完毕之后，最好把它设回原先的默认值STDOUT [注44]。将数据输出到文件句柄时，默认情况下都会经过缓冲处理。不过，只要将特殊变量$|设定为1，就会使当前的（也就是修改变量时所指定的）默认文件句柄在每次进行输出操作后立刻刷新缓冲区。所以，如果要让输出的内容立即显示（比如在读取监视某个耗时程序的实时日志时），应该这么做：

```
select LOG;
$| = 1;  # 不要将 LOG 的内容保留在缓冲区
select STDOUT;
# ……时间流逝，婴儿都学会走路了，斗转星移之后……
print LOG "This gets written to the LOG at once!\n";
```

重新打开标准文件句柄

我们之前提过，如果重新打开某个文件句柄（比如要打开某个名为FRED的文件句柄时，已经有一个处于打开状态的同样名为FRED的文件句柄），Perl会自动帮你关闭原来那个。我们也提到过，你不应该复用Perl的6个标准文件句柄，除非你想使用该文件句柄实现特殊功能。我们还说过，来自die和warn的信息以及Perl内部的出错信息都会自动送到STDERR。如果以上三个知识能融会贯通，你就会意识到，错误信息不一定都要送到程序的标准错误输出流，也可以送到文件里 [注45]：

注44： 在某些情况下可能想选择的句柄不是STDOUT，你可以在*perlfunc*文档中查看关于select的内容，了解如何保存和恢复当前文件句柄。另外，实际上在*perlfunc*文档中你能找到两个Perl内置的select函数，它们虽然名字一样，但另一个select在调用时需要提供4个参数。

注45： 如果没有充分理由的话，请不要这样做。最好是让用户自己在运行程序时决定什么信息该重定向到哪里，而不是由你把它硬性规定在程序里。但是当你的程序被其他程序自动运行（比如说被Web服务器或者cron或at这些进程调度工具程序）时，又或者你的程序要启动其他进程（可能是用system或exec函数，你会在第十四章看到）并且需要这个进程有不同的I/O时，这样做会比较灵活。

```
#将错误信息写到我自己的错误日志中
if ( ! open STDERR, ">>/home/barney/.error_log") {
    die "Can't open error log for append: $!";
}
```

在重新打开了STDERR之后，任何从Perl产生的错误信息都会送到新的文件里。但如果程序执行到die这部分的代码，那会怎样呢？也就是说，如果无法成功打开文件来接收错误信息，那么错误信息会流到哪里去？

答案是：在重新打开这三个系统文件句柄STDIN、STDOUT或STDERR失败时，Perl会热心地帮你找回原先的文件句柄 [注46] 。也就是说，Perl只有在成功打开新的句柄连接时，才会关闭默认的系统文件句柄。所以程序可以用这个技巧对三个系统文件句柄中任何一个（或全部）进行重定向 [注47] ，这跟“程序从命令行运行时就由shell进行I/O重定向”的功能是一样的。

用say来输出

Perl 5.10从正在开发的Perl 6中借来了say这个函数（而Perl 6中的say函数可能是借鉴了Pascal的println函数）。它的功能和print函数差不多，但在打印每行内容时会自动加上换行符。所以下面这几种写法的最终输出结果都一样：

```
use 5.010;

print "Hello!\n";
print "Hello!", "\n";
say "Hello!";
```

如果只是要打印某个变量值并在末尾附带换行符，其实不必额外构造字符串，也不必给print函数提供数据列表，只要直接say这个变量就可以了。在你想输出某些内容并换行时，这个函数非常好用：

```
uuse 5.010;

my $name = 'Fred';
print "$name\n";
print $name, "\n";
say $name;
```

但在内插数组时，最好还是用引号将它括起来，以便用空格隔开数组中的每个元素：

注46： 至少在你没修改特殊变量$^F的情况下，Perl会这么做。这个特殊变量告诉Perl在复用这三个句柄失败的时候恢复它们的默认值，最好不要修改该变量。

注47： 别把STDIN修改成用来输出的文件句柄，或者把另外两个标准输出句柄改成用来输入的句柄。这种把它们搞混淆的事情，想想都头痛。

```
use 5.010;

my @array = qw( a b c d );
say @array;   # 打印 "abcd\n"
say "@array"; # 打印 "a b c d\n";
```

和print函数一样，你可以为say指定一个文件句柄：

```
use 5.010;

say BEDROCK "Hello!";
```

因为这是Perl 5.10的新特性，所以只能在启用Perl 5.10新特性的情况下使用。虽然传统的print一样可以工作得很好，不过我们觉得一定有许多Perl程序员会马上换用这个新函数，这样每次都可以省掉4次按键（函数名可以节约两个，再加上\n这两个字符）。

标量变量中的文件句柄

从Perl 5.6开始，我们已经可以把文件句柄存放到标量变量中，而不必非得使用裸字。别小看这点差别，带来的好处可不少。成为标量变量后，文件句柄就可以作为子程序的参数传递，或者放在数组、哈希中排序，或者严格控制它的作用域。当然，有关裸字的用法还是需要谨记在心的，很多时候我们写的都是应急的短小脚本，用裸字更快捷，没必要用变量存储文件句柄。

在open函数中原来使用裸字的地方写上不含有任何值的变量，那么文件句柄就会存放到那个变量中。人们一般都会使用词法变量以确保该变量预先是空的，有些人喜欢在变量名后面添上_fh表示这是用来保存文件句柄的变量：

```
my $rocks_fh;
open $rocks_fh, '<', 'rocks.txt'
    or die "Could not open rocks.txt: $!";
```

甚至于你还可以把这两步并作一步，直接在open函数中声明词法变量：

```
open my $rocks_fh, '<', 'rocks.txt'
    or die "Could not open rocks.txt: $!";
```

得到保存文件句柄的变量之后，只要把原来使用裸字的地方改成用这个变量就可以了：

```
while( <$rocks_fh> ) {
    chomp;
    ...
}
```

输出信息到某个文件句柄也可以使用这个变量。在原来使用裸字的地方使用这个标量变

量，便能以适当的模式打开该文件句柄：

```
open my $rocks_fh, '>>', 'rocks.txt'
    or die "Could not open rocks.txt: $!";

foreach my $rock ( qw( slate lava granite ) ) {
    say $rocks_fh $rock
}

print $rocks_fh "limestone\n";
close $rocks_fh;
```

请注意，这种写法仍然不需要额外的逗号。Perl能够自动判断，如果跟在`print`后面的第一个参数之后没有逗号，就说明它是一个文件句柄，即此处的`$fh`。要是误加了逗号，那么打印出来的东西会看起来怪怪的，这当然不是你所希望的：

```
print $rocks_fh, "limestone\n"; # 错误
```

这段代码会输出类似下面这样的字符串：

```
GLOB(0xABCDEF12)limestone
```

这是怎么回事？说起来也很简单，因为第一个参数后出现了一个逗号，所以Perl会把这个参数当作要打印的字符串而不是文件句柄来处理。虽然我们还没介绍过有关引用的概念，不过如果有兴趣，可以翻翻我们另外的进阶书籍《Intermediate Perl》，其中有关将引用字符串化（*stringification*）的内容会详细解释内在机理。所以，根据上面的原则，下面这两条语句实质上是不同的：

```
print STDOUT;
print $rock_fh;  # 错误，这应该不是你的本意
```

在第一个例子中，Perl知道`STDOUT`是文件句柄，因为它就是个裸字。由于其后没有任何参数，所以它会打印默认变量`$_`中的内容。而在第二个例子中，Perl无法预先判断`$rock_fh`是否为文件句柄，只有运行到这条语句时才知道变量里面保存的是不是文件句柄，所以它只好假设标量变量`$rock_fh`是要输出的字符串变量。要解决这样的问题并不难，只要用花括号围住文件句柄，Perl就能明白它的正确含义，即便这个文件句柄保存在数组或哈希中也没关系：

```
print { $rock_fh }; # 默认打印$_中的内容
print { $rocks[0] } "sandstone\n";
```

根据实际编程时的情况，选择使用裸字或者标量变量都没问题。一般短小的程序，比如系统管理员的工具脚本，用用裸字也没什么不好。不过对大一点的项目或应用程序来说，使用标量标量的方式可以精确控制文件句柄的作用域，方便调试和维护。

习题

以下习题答案参见第314页上的“第五章习题解答”一节：

1. [7]写一个功能跟*cat*相似的程序，但将各行内容反序后输出（有些操作系统会有一个名为*tac*的类似工具）。假如用`./tac fred barney betty`来运行你的程序，它的输出结果应该是*betty*文件的最后一行到第一行，接着是文件*barney*与*fred*，同样是由最后一行到第一行。（如果你将此程序取名为*tac*，请一定要在运行时加上`./`，这样才不会运行你的系统中现有的同名程序！）

2. [8]写一个程序，要求用户分行键入各个字符串，然后以20个字符宽、向右对齐的方式输出每个字符串。为了确定输出结果在适当的字段中，请一并输出由数字组成的“标尺行(rule line)”（只是为了方便调试）。请确定自己没有误用19个字符宽的字段！比如输入`hello`、`good-bye`后应该会得到下面这样的输出结果：

```
1234567890123456789012345678901234567890123456789012345678901234567890
               hello
            good-bye
```

3. [8]修改上一个程序，让用户自行选择字符宽度，因此在键入30的时候，`hello`、`good-bye`（在不同行上）应该会向右对齐到第30个字符（提示：关于如何控制变量的内插，请参阅第二章中的“字符串中的标量变量内插”一节）。附加题：根据用户键入的宽度，自动调整标尺行的宽度。

第六章

哈希

本章我们会看到Perl成为杰出编程语言的关键特色——哈希[注1]。尽管哈希非常强大有用，但那些多年使用其他语言的人却可能从未听说。不过从现在开始，几乎在每个Perl程序中你都会用到哈希。是的，它真的非常重要。

什么是哈希？

哈希是一种数据结构，它和数组的相似之处在于可以容纳任意多的值并能按需取用，而它和数组的不同在于索引方式，数组是以数字来索引，哈希则以名字来索引。也就是说，哈希的索引值，此处称为键(key)，并不是数字，而是任意唯一的字符串（参见图6-1）。

哈希的键其实就是字符串，所以打个比方，我们不必用数字3来获取数组元素，我们可以用wilma这个名字来存访问希元素。

这些键可以是任何字符串——你可以用任意字符串表达式作为哈希键。它们也必须是唯一的字符串，就像数组只能有一个编号为3的元素一样，哈希也只能有一个名叫wilma的元素。

另一种看待哈希的方法，是将它想象成一大桶数据，其中每个数据都有关联的标签。你可以伸手到桶里任意取出一张标签，看它上面附着的数据是什么。但是桶里没有所谓的“第一个”元素，只有一堆数据。在数组里，第一个元素为元素0，然后是元素1、2等等，但哈希里没有顺序，因此也没有所谓的第一个元素，有的只是一堆键-值对的集合罢了。

注1：以前我们是叫它“关联数组（associative array）”的。但在1995年前后，Perl社区认为这个名字太啰嗦，写起来也麻烦，于是我们把名字改成了“哈希（hash）”。

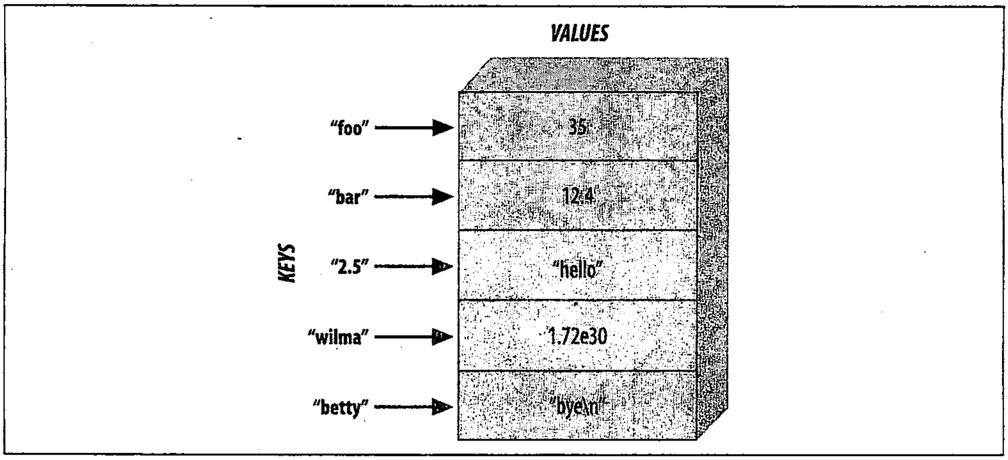


图6-1：哈希的键与值

这些键和值都是任意的标量，但键总会被转换成字符串。假如你以数字表达式50/20为键[注2]，那么它会被转换成一个含有三个字符的字符串"2.5"，恰好就是图6-2中所示的一个键。

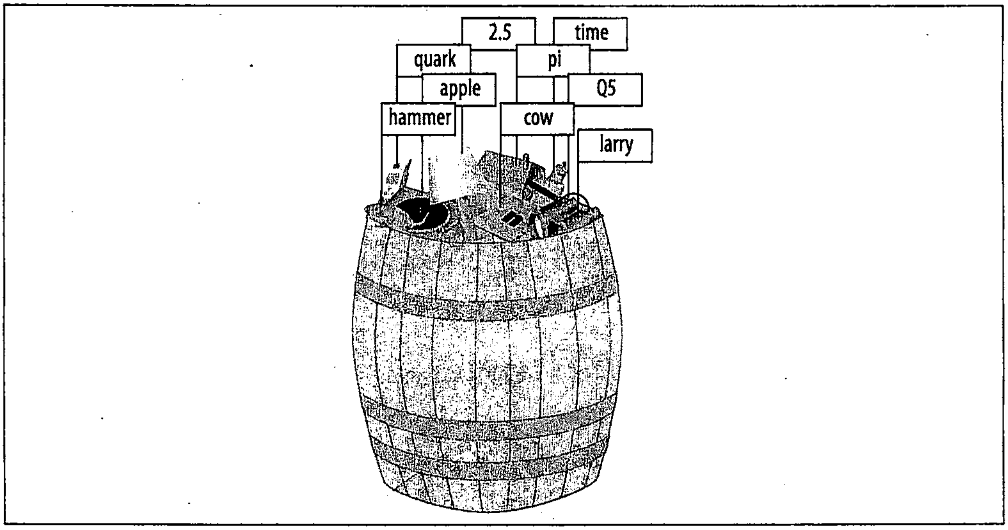


图6-2：哈希像一桶数据

和以往一样，根据Perl“去除不必要的限制”的原则，哈希可能是任意大小的，从没有任何键-值对的空哈希到填满内存的巨大哈希都可以。

注2：　这是一个数字表达式，而不是有5个字母的字符串"50/20"。当然如果我们使用了该字符串作为哈希键，那它就会保持不变。

某些语言的哈希实现在键-值对增多时会逐渐变慢，例如*awk*语言就是如此。Larry正是从这种语言中引入了哈希。但Perl的版本没有这个问题，它有良好、高效、可伸缩的算法[注3]。因此，如果某个哈希只有三个键-值对，那么从中提取任意一项数据都会非常快捷。如果某个哈希包含300万个键-值对，从中提取任意一项数据还是会和原来一样快，所以不用担心巨硕的哈希的读写性能。

值得一提的是：虽然这些键是唯一的，但它们对应的值是可以重复的。哈希的值可以是数字、字符串、`undef`，或是这些类型的组合[注4]。但哈希的键则必须全部是唯一的字符串。

为何使用哈希？

当你第一次听到哈希时，特别是在你以前用得很好的程序语言里没有哈希时，也许会对为何有人需要这个奇特的怪物而感到困惑。其实这个想法的根源就是你总是需要将一组数据“对应到”另一组数据。例如下面这些典型的应用场景：

按名字找姓

以名字作为键，而姓可以成为相应的值。这当然需要限定名字是唯一的，如果出现了两个叫做`randal`的人就行不通了。通过哈希可以按任何人的名字找到相应的姓。例如以`tom`为键可取得值`phoenix`。

按主机名找IP地址

你也许知道在因特网上每台计算机同时拥有一个主机名（比如*http://www.stonehenge.com*）以及一个IP地址（比如123.45.67.89）。这是因为机器喜欢和数字打交道，但是一般人比较记得住名字。主机名是唯一的字符串，可以用来作为哈希键。通过哈希可以按主机名找到相应的IP地址[注5]。

按IP地址找主机名

当然，你也可以反其道而行。我们一般把IP地址当成数字，但它们也是唯一的字符串，因此可以成为哈希键。在这个哈希中，我们可以查询IP地址以决定相应的主机名。请注意，这个哈希并不等同于上面例子中的哈希：哈希是从键到值的单行道，我们无法在哈希中查询值并反推出其相应的键！所以这两个例子是成对的哈希，一

注3：　技术上来说，Perl会在需要时重建大型哈希表。其实之所以用“哈希”这个术语就是因为数据类型就是用哈希表实现的。

注4：　事实上，任何标量值都可以，也包括目前还没提到的其他标量类型。

注5：　其实这个例子并不恰当，我们知道有些主机名其实可以拥有多个IP地址，而同一个IP地址上也可以绑定多个主机名。不过这里只是说明下概念，你明白就可以了。

个存放IP地址，另一个存放主机名。然而有了其中一个哈希，要生成另一个哈希是相当容易的，稍后可以看到。

按单词统计其出现次数

这是个极为常见的哈希应用，因为相当常见，所以可能会在章末的习题里面出现。

这里的想法是要知道某个文件里每一个单词出现的频率。也许你正在为一堆文件编写索引，然后当用户检索`fred`的时候，就能知道这个词在甲文件中出现了5次，在乙文件中出现了7次，而在丙文件中则从未出现。有了这个索引我们就知道哪个文件可能是用户最感兴趣的。在索引构造程序扫描每份指定的文件时，对于每个找到的`fred`，就给`fred`这个键对应的值加1。也就是说，假如`fred`之前在文件中已经出现过两次，它的值就是2，现在它会被加到3。假如我们以前未曾见过`fred`，该值就会从默认的`undef`加到1。

按用户名统计每个人使用（或者浪费）的磁盘块数量

系统管理员会喜欢这个例子：系统中的用户名是唯一的字符串，因此可以被当成哈希的键来检索与用户相关的信息。

按驾驶执照号码找出姓名

也许有许多人都叫做John Smith，但是起码他们应该有不同的驾照号码。因此唯一的驾照号码可以成为键，而人名可以作为值。

另一种思考方式是将哈希当成极其简单的数据库，其中每个键的“名下”都只能存储一块相应的数据。事实上只要问题中带有“找出重复”、“唯一”、“交叉引用”、“查表”之类的字眼，实现时就很有可能会用到哈希。

访问哈希元素

要访问哈希元素，需要使用如下语法：

```
$hash{$some_key}
```

这和访问数组的做法类似，只是使用了花括号（curly braces）而非方括号（square brackets）来表示索引值(哈希键) [注6] 。并且，现在的键表达式是字符串，而非数字：

```
$family_name{'fred'}   = 'flintstone';
$family_name{'barney'} = 'rubble';
```

图6-3显示了如何对哈希键赋值。

注6：　为什么要用花括号，而不用方括号呢？Larry的解释是：因为哈希的访问方法要比常规的数组访问更酷一些，所以也自然需要使用更花哨的括号。

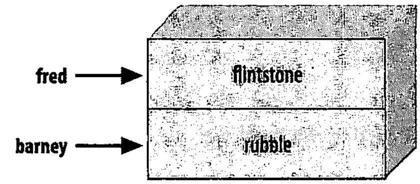


图6-3：哈希键的赋值

这就允许我们写出这样的代码：

```
foreach my $person (qw< barney fred >) {
  print "I've heard of $person $family_name{$person}.\n";
}
```

哈希变量的命名和其他Perl的标识符相似，可以有字母、数字和下划线，但不可以用数字开头。另外哈希有自己的名字空间，也就是说哈希元素$family_name{"fred"}和子程序&family_name之间毫无关联。当然也没有必要故弄玄虚地把所有东西都起同样的名字。假如同时有一个名叫$family_name的标量变量以及像$family_name[5]这样的数组元素，Perl也毫不在意。我们人类要学习Perl的做法，换句话说，我们必须仔细看清楚标识符前后的标点符号来判断它的真实意义。倘若名称之前有一个美元符号而之后紧接着花括号，那么此处访问的就是一个哈希元素。

在挑选哈希名的时候，最好使得哈希名和键之间能放进去一个“for”字。比如“family_name for fred是flintstone”，因此把哈希命名为family_name能清晰地反映出键和值之间的关系。

当然，哈希键不一定是上面例子里的字符串或简单的标量变量，也可以是任意的表达式：

```
$foo = 'bar';
print $family_name{ $foo . 'ney' };  # 打印 "rubble"
```

若对某个已存在的哈希元素赋值，就会覆盖之前的值：

```
$family_name{'fred'} = 'astaire';  # 给已有的元素赋上新值
$bedrock = $family_name{'fred'};   # 得到 "astaire"，早先的值已不存在
```

这和数组与标量的情形类似，如果对$pebbles[17]或者$dino赋值，就会覆盖其之前的值。如果对$family_name{"fred"}赋值，同样会覆盖其之前的值。

哈希元素会因赋值而诞生：

```
$family_name{'wilma'} = 'flintstone';            # 增加一个新的键-值对
```

```
$family_name{'betty'} .= $family_name{'barney'};  # 在需要的时候动态创建该元素
```

这和数组与标量的情形如出一辙[注7]，假如以前不存在$pebbles[17]或$dino，赋值之后这些变量就会出现；假如以前不存在$family_name{"betty"}，现在就可以如法炮制。

访问哈希表里不存在的值会得到undef：

```
$granite = $family_name{'larry'};  # 没有larry这个键，所以值为undef
```

是的，这个效果跟数组与标量的情形相似：假如$pebbles[17]或$dino里还没有值，访问这些变量时会得到undef；假如没有任何值存放在$family_name{"larry"}中，访问它的时候也会得到undef。

访问整个哈希

要指代整个哈希，可以用百分号（%）作为前缀。因此前面我们使用的哈希准确来说应该称之为%family_name。

为了方便起见，哈希可以被转换成列表，反之亦然。对哈希赋值（下面这个例子来自于图6-1）等同于在列表上下文中赋值，列表中的元素应该为键-值对[注8]：

```
%some_hash = ('foo', 35, 'bar', 12.4, 2.5, 'hello',
      'wilma', 1.72e30, 'betty', "bye\n");
```

在列表上下文中，哈希的值是简单的键-值对列表：

```
@any_array = %some_hash;
```

我们把这个变换叫做展开（*unwinding*）哈希，将它变成键-值对列表。当然，得到的键-值对不一定是按照当初赋值时的顺序展开：

```
print "@any_array\n";
  # 可能会给出像这样的结果：
  # betty bye（以及一个换行符）wilma 1.72e+30 foo 35 2.5 hello bar 12.4
```

之所以顺序乱掉是因为Perl已经为哈希的快速检索而对键-值对的存储作了特别的排序[注9]。因此选择使用哈希的场合，要么元素存储顺序无关紧要，要么可以容易地在元素输出时进行排序。

注7：这种特性称为自动延展（*autovivification*），我们在《Intermediate Perl》一书中有讨论。

注8：尽管任何列表都可以使用，但必须得有偶数个元素，因为哈希必须是由键-值对组成。奇数会导致不可靠的结果，当然你仍然可以忽略这个警告。

注9：Perl还会刻意打乱键-值对的顺序，这样黑客就无法预知信息是如何存储的。

当然，即使键-值对的顺序被打乱，列表里的每个键还是会“黏着”相应的值。所以，即使无法知道某个键foo会出现在列表的哪个位置，仍然可以确信相应的值35会跟在后面。

哈希赋值

这不是常见用法，但哈希真的可以用一般的赋值语法来复制：

```
my %new_hash = %old_hash;
```

这里Perl做的工作要比看到的繁杂得多。不像Pascal或C语言里简单的复制内存块的做法，Perl的数据结构更为复杂，所以底层实现也大不相同。大致上，这行代码会先把%old_hash展开为键-值对列表，然后通过列表赋值重新构造每个键-值对，最终形成新的哈希%new_hash。

但根据现有哈希转换得到新哈希倒是非常常见。比如建立一个反序的哈希：

```
my %inverse_hash = reverse %any_hash;
```

这会将%any_hash展开成为键-值对列表，看起来是（*key,value,key,value,key,value,…*)这样。然后利用reverse的列表翻转功能形成一个（*value,key,value,key,value,key,…*）这样的新列表，键值达成互换。当结果存回%inverse_hash时，我们就能以原本在%any_hash里的值来进行检索，它已经成为%inverse_hash的键，而按它找到的值则是%any_hash的某个键。现在我们能够按“值”(现在是键)来找“键”(现在是值)了。

当然敏锐的读者可能猜到这种技巧只能在哈希值不重复的情况下才能奏效——否则就会导致重复的键，而这对于哈希是不允许的。对于这个问题Perl采用“后发先至”的原则，用列表中最后的键覆盖之前的键。

当然我们说过了键-值对在展开哈希之后的顺序是无法预知的，所以也无法预知哪个键才会被覆盖。这个技巧最好是在确定原始哈希的值是唯一的情况下使用[注10]。这个技巧其实非常适合前面提到的IP地址和主机名的例子：

```
%ip_address = reverse %host_name;
```

这样一来，我们就可以轻松地用主机名或IP地址来检索相应的IP地址或主机名了。

注10：　或是当你不介意哈希键重复时。比如我们可以将%family_name哈希从名-姓表反转成为姓-名表，用来查看某个姓是否存在。如果反转后的哈希没有slate键，我们就能确定原始的哈希里没有是这个姓的成员。

胖箭头

在将列表赋值到哈希时常常会发现列表中的键-值对并不容易区分。比如在下面的赋值中，任何人都要逐个扫描列表成员，同时默念着："键、值，键、值……"，然后才搞清楚2.5其实是一个键，而不是值：

```
%some_hash = ('foo', 35, 'bar', 12.4, 2.5, 'hello',
      'wilma', 1.72e30, 'betty', "bye\n");
```

如果Perl能让我们将此类列表中的键与值成对组合，方便区别谁是谁，岂不更好？Larry也有感于此，因此他发明了胖箭头（=>）[注11]。对Perl而言，它只是逗号的另一种写法，因此我们常常称呼它为胖逗号。也就是说，在任何需要逗号（,）的地方都可以用胖箭头代替，这对Perl来说没什么区别[注12]。所以产生名-姓哈希的另一种方式是：

```
my %last_name = (  # 哈希也可以是词法变量
  'fred'   => 'flintstone',
  'dino'   => undef,
  'barney' => 'rubble',
  'betty'  => 'rubble',
);
```

这样组合的姓和名看起来更加清晰了，哪怕把所有的姓和名都写在一行里面也可以读懂。请注意，列表结尾有一个额外的逗号，这种写法不但无伤大雅，而且便于维护。当需要加入更多人的信息的时候，只要确保每行都有一组键-值对和结尾的逗号就行了。Perl会明白每个键-值对之间隔着逗号，整个列表以一个额外的(无伤大雅的)逗号结尾。

这样已经好很多了。Perl总是会竭尽所能让程序员使用简写，这里就有一个：使用胖箭头的时候可以省略键的引号，左边的部分会被自动引起：

```
my %last_name = (
  fred   => 'flintstone',
  dino   => undef,
  barney => 'rubble',
  betty  => 'rubble',
);
```

当然，也不是所有情况都可以这么做，因为哈希的键可以是任意形式的字符串，所以要

注11：　没错，还有一个瘦箭头->，它是用于引用（reference）的，可以参考*perlreftut*或者 *perlref*文档了解引用这个高级主题。

注12：　哦，其实还有一个细微的差别：胖箭头左边的任何裸字(一连串的字母、数字和下划线，但不得以数字开头)都会自动加上引号，因此胖箭头左边的裸字不需要加引号。另外，在作为哈希键使用时，如果花括号内只有裸字时，则两边的引号也可以省略。

是某个键的内容看起来像是Perl的操作符的话，Perl就无法适从了。比如下面放在胖箭头左边的+是加法操作符，不是用引号引起的字符串：

```
my %last_name = (
  +   => 'flintstone',  # 错了！编译错误！
);
```

不过一般来讲，键都是非常简单的字符串，如果键名只是由字母、数字和下划线组成的，并且不是以数字开头，那就可以省略引号。这类无需引号的字符序列，我们称之为裸字（bareword），因为它是孤立存在的。

还有一个常见的允许省略键名引号的地方是：在花括号中检索特定键名的元素。比如原来的$score{'fred'}可以直接简写为$score{fred}。由于许多哈希键名都是这类简单的单词，所以不加引号的写法几乎成了惯例。但要注意，如果花括号内不是裸字，Perl就会将其当作表达式先求值，然后把结果当作键名。比如，Perl会认为下面的是字符串连接操作：

```
$hash{ bar.foo } = 1;  # 构成键名 'barfoo'
```

哈希函数

很自然，Perl有很多有用的函数可以一次处理整个哈希。

keys和values函数

keys函数能返回哈希的键列表，而values函数能返回对应的值列表。如果哈希没有任何成员，则两个函数都返回空列表：

```
my %hash = ('a' => 1, 'b' => 2, 'c' => 3);
my @k = keys %hash;
my @v = values %hash;
```

所以，@k会包含'a'、'b'和'c'，而@v则会包含1、2和3。当然顺序会有所不同，别忘了不能预测Perl存储哈希的顺序。但可以确定的是，返回的键列表和值列表的顺序是一致的：如果'b'是键列表的最后一个元素，那么2也一定是值列表的最后一个元素；如果'c'是键列表的第一个元素，那么3也一定是值列表的第一个元素。只要在取得键与取得值这两个动作之间不修改哈希，顺序必然一致。但如果新增某个元素的话，Perl就可能根据需要重新优化排列顺序，以保持后续高速访问[注13]。在标量上下文中，这两个函数都

注13： 当然，如果你在keys和values调用之间又增加了新的哈希元素的话，那么返回的两个列表就会有不同数量的元素，自然也就无法匹配。所以正常情况下没人会这么做。

会返回哈希中元素(键-值对)的个数。这个计算过程不必对整个哈希进行遍历，因而非常高效：

```
my $count = keys %hash;  # 得到 3，也就是说有三对键值
```

偶尔也能看到别人的程序里把哈希当成布尔表达式来判断真假，比如：

```
if (%hash) {
  print "That was a true value!\n";
}
```

只要哈希中至少有一个键-值对，就返回真[注14]。所以，这样写的意思就是：假如哈希不是空的则应该如何如何。不过，这种写法并不多见。

each函数

如果需要迭代(逐项处理其中的每一个元素)整个哈希，常见的写法就是用each函数，它以包含两个元素的列表的形式返回键-值对[注15]。每次对同一个哈希调用此函数，它就会返回下一组键-值对，直到所有的元素都被访问过。在没有任何新的键-值对，此时each会返回空列表。

实际使用时，唯一适合使用each的地方就是在while循环中，如下所示：

```
while ( ($key, $value) = each %hash ) {
  print "$key => $value\n";
}
```

这里有很多技巧。首先，each%hash会从哈希中返回一组键-值对，结果是含有两个元素的列表：如果键是"c"而值是3，则列表就会是("c",3)。该列表会被赋值给($key,$value)，因此$key会成为"c"，而$value则变成3。

但这里的列表赋值操作是在while循环的条件表达式中发生的，也就是在标量上下文中赋值(说得再具体些，最终在while内部的是布尔上下文，目的是要求真假值，而布尔上下文是一种比较特殊的标量上下文)。赋值后得到的列表在标量上下文中的求值结果为列表的元素数量，所以在这个例子中，我们得到的是2。因为2是真值，所以继续运行循环块并打印c=>3。

注14：实际结果是对Perl维护人员很有用的一个内部调试用的字符串。字符串类似于“4/16”这样的形式，如果哈希不是空的，这个值就是真；若是空哈希，则返回假。因此普通用户可以把它当成布尔值使用。

注15：另一个迭代整个哈希的常见方法是用foreach遍历哈希的键列表，这一节末尾能看到这种写法。

下一次循环中，each%hash会返回一组新的键-值对，假设这次是("a",1)。之所以能返回下一个键-值对，是因为哈希还记着上次访问的位置，用技术行话来说就是每个哈希都有一个迭代器（iterator）[注16]。这两个值会被存进($key,$value)。因为列表的元素个数还是2，亦即真值，所以继续运行循环块，输出a=>1。

再一次运行循环时，我们已经知道会发生什么事了，所以看到输出是b=>2时不会意外。

我们知道循环不可能一直运行下去。当Perl执行each%hash却已经没有任何键-值对时，each会返回空列表[注17]。空列表会被赋值到($key,$value)，因此$key得到undef，$value也会得到undef。

但因为这是在while循环的条件表达式中运算，所以刚才的赋值都不重要。在标量上下文中，列表赋值运算的值是源列表中元素的个数，在此情况下是0。因为0这个值为假，所以while循环就结束了，会继续运行程序接下来的部分。

当然，each返回键-值对的顺序是乱的。但它与keys和values返回的顺序相同，也就是哈希的自然顺序。假如你需要依次处理哈希，只要对键排序就行了。方法如下所示：

```
foreach $key (sort keys %hash) {
  $value = $hash{$key};
  print "$key => $value\n";
  # 或者，我们也可以略去额外的 $value 变量：
  #  print "$key => $hash{$key}\n";
}
```

我们将会在第十四章看到更多有关哈希排序的内容。

哈希的典型应用

讲到这里，不妨一起来看看哈希的实际应用。

Bedrock图书馆使用了一个Perl程序，通过一个哈希来记录每个人当前借走几本书，当然除此之外还有其他相关信息：

注16：由于每个哈希有自己的迭代器，因此处理不同哈希的each调用可以嵌套。既然大家已经习惯了本书的脚注风格，我们不妨为你介绍一些少见的技巧：使用keys或者values函数可以重置哈希的迭代器。另外用新列表重置整个哈希时也可以重置迭代器，或者each调用遍历了整个哈希的时候也能重置迭代器。然而在迭代哈希过程中增加新的键-值对就不太好，因为这不会重置迭代器，反而会迷惑开发人员、程序维护员，另外还会愚弄each。

注17：因为是在列表上下文中使用，所以它不能返回undef表示失败，那会成为拥有一个元素的列表(undef)而不是空列表()。

```
$books{'fred'} = 3;
$books{'wilma'} = 1;
```

要判断某项哈希元素的真假很简单，只要这么做：

```
if ($books{$someone}) {
  print "$someone has at least one book checked out.\n";
}
```

不过哈希里有些元素并不为真：

```
$books{"barney"}  = 0;      # 现在没有借阅图书
$books{"pebbles"} = undef;  # 从未借阅过图书，这是张新办的借书证
```

因为Pebbles不曾借过任何书，所以她的借出数量是undef，而不是0。

每个有借书证的人在哈希里都有相应的键。对于每个键（也就是图书馆的借阅者）来说，它都有相应的值，这个值若不是借出图书的数量，就是undef——代表着一个从未使用过借书证的读者。

exists函数

若要检查哈希中是否存在某个键（也就是某人是否有借书证），可以使用exists函数，它能返回真或假，分别表示键存在与否，和键对应的值无关：

```
if (exists $books{"dino"}) {
  print "Hey, there's a library card for dino!\n";
}
```

也就是说，exists$books{"dino"}会返回真，如果(且仅如果)dino存在于keys%books返回的键列表中的话。

delete函数

delete函数能从哈希中删除指定的键及其相对应的值。假如没有这样的键，它就会直接结束，而不会出现任何警告或错误信息。

```
my $person = "betty";
delete $books{$person};  # 撤销$person的借书证
```

请注意，这与“将undef存入哈希元素”并不相同。在这两种情况下，exists($books{"betty"})会得出相反的结果。在delete之后，键便不会出现在哈希之中，但存入undef后，键却是一定会存在的。

在这个例子中，delete与存入undef的差异，就如同拿走Betty的借书证与给她一张没用过的借书证一样完全不同。

哈希元素内插

可以将单一哈希元素内插到双引号引起的字符串中，就和你想要的一样：

```
foreach $person (sort keys %books) {              # 按次序访问每位借阅者
  if ($books{$person}) {
    print "$person has $books{$person} items\n";   # fred 借了 3 本书
  }
}
```

但这种方式不支持内插整个哈希，"%books"的意思只是包含6个字符的字符串%books [注18]。到这里为止，我们已经看到了所有在双引号中需要反斜线转义的魔力字符：$和@，因为它们引入一个Perl将要内插的变量；"，若不用反斜线转义，这个符号就会结束双引号引起的字符串；\，代表反斜线本身。除这些以外，双引号中任何字符都只代表他们自己，无需转义 [注19]。

%ENV哈希

Perl程序既然运行在某个环境(*environment*)中，就需要对周围的环境有所感知。Perl访问这些信息的方法是访问%ENV哈希。比如，我们常常需要从%ENV中读取PATH键的值：

```
print "PATH is $ENV{PATH}\n";
```

根据你所使用的操作系统和设定，大致会看到类似下面这样的输出：

```
PATH is /usr/local/bin:/usr/bin:/sbin:/usr/sbin
```

一般这些环境变量都早已自动设置好，但你也可以添加自己的环境变量。不同的操作系统和shell有不同的设定方法：

注18：它实在没法代表任何有意义的东西，如果想用它输出哈希里所有键-值对，那么这堆信息实际上也没太大用处。并且，我们在第五章中看到过，百分比号经常会出现在printf的格式字符串里，如果再赋予它其他意义，准会变得一团糟。

注19：但在双引号引起的字符串中小心变量名之后的缩写符号（'）、左方括号（[）、左花括号（{）、瘦箭头（->）和双冒号（::），因为它们可能会带来与你本意相悖的效果。

Bourne shell

```
$ CHARACTER=Fred; export CHARACTER
$ export CHARACTER=Fred
```

csh

```
% setenv CHARACTER Fred
```

DOS或者Windows命令

```
C:> set CHARACTER=Fred
```

只要在程序外设定任意环境变量，在Perl里面就可以这样访问：

```
print "CHARACTER is $ENV{CHARACTER}\n";
```

习题

以下习题答案参见第316页上的“第六章习题解答”一节：

1. [7]编程读入用户指定的名字并且汇报相应的姓。拿熟人的姓和名测试，如果你太专注于计算机以至于一个人也不认识的话，也可以使用下面这个列表：

   表6-1：数据样本

输入	输出
fred	flintstone
barney	rubble
wilma	flintstone

2. [15]编程读取一系列单词，每行一个[注20]，直到文件中止，然后打印一份列出每个单词出现次数的列表。（提示：别忘了，把未定义值当成数字使用时，Perl会自动将它转换成0。回头去看看前面计算总和的习题，可能会有所帮助。）这样，如果输入单词为fred、barney、fred、dino、wilma、fred，每个词一行，输出应该告诉我们fred出现了3次。附加题：根据ASCII编码排序输出报表。

3. [15]编程输出%ENV哈希中所有的键-值对，输出按照ASCII编码排序，分两列打印。附加题：设法让打印结果纵向对齐。注意length函数可以帮助确定第一列的宽度。测试完毕后加入更多新环境变量再次验证程序的输出正确无误。

注20：必须每个单词分行输入，因为我们暂时还没有介绍如何对一行输入进行分词处理。

第七章

漫游正则表达式王国

Perl有众多区别于其他语言的特色。在这些特色中最重要的就是对正则表达式的强力支持。这些支持提供了快速、灵活、可靠的字符串处理能力。

不过，这个能力是有代价的。正则表达式其实是Perl内嵌的、自成一体的微型编程语言。没错，你正在学习另一门语言[注1]！好在这个语言并不难。在这一章，你将会进入正则表达式的王国，暂时忘掉Perl世界也没关系。在下一章我们会告诉你，这个王国是如何融入Perl世界的。

正则表达式并不只是Perl的一部分，它也出现在*sed*、*awk*、*procmail*与*grep*中，以及大多数程序员文本编辑器（比如*vi*和*emacs*）中，甚至是更奇怪的地方。好消息是，如果你用过上述某些工具，那么已经赢在起点了。再仔细看看，你会发现更多使用或支持正则表达式的工具，像Web上的搜索引擎（通常就是用Perl写成的）、电子邮件客户端等等。坏消息是各家正则表达式的语法不尽相同，在学习过程中，可能还得习惯时不时出现的反斜线。

什么是正则表达式？

正则表达式（*regular expression*），在Perl里面通常也叫做模式（*pattern*），是用来表示匹配（或不匹配）某个字符串的特征模板[注2]。也就是说，虽然有无限多可能的文

注1：　可能会有人说正则表达式不是完整的编程语言。我们懒得辩解，不管怎么说，Perl确实有在正则表达式内嵌入额外 Perl 代码的能力。

注2：　爱较真的人会要求更严谨的定义。可是即使给出定义，他们也会说Perl的模式不算真正的正则表达式。你如果想认真学习正则表达式，我们推荐 Jeffrey Friedl写的《Mastering Regular Expressions》（O'Reilly 出版）一书。

本字符串存在，但只要用一个模式就可以将它们干净利落地分成两组：匹配的和不匹配的。模式绝对没有仁慈、写意之类的性格，它要么匹配，要么就不匹配。

模式可能只匹配一个给定的字符，或者两个、三个、十个、上百个，甚至无数个字符。当然，模式也可以匹配所有除了给定的一个、多个或无限多个字符以外的内容[注3]。前面说了，正则表达式是一种小程序，它拥有自己的简单编程语言。其实这个程序的任务很简单：查看一个字符串，然后判定它“匹配”或“不匹配”[注4]。这就是它所做的全部工作。

另外一个用到正则表达式的地方就是Unix的*grep*命令，它会检查哪几行文本匹配指定的模式，然后输出那几行。比如，想看看某个文件在哪一行提到过flint，并且同一行内还跟着stone，可以用下面这条*grep*命令：

```
$ grep 'flint.*stone' chapter*.txt
chapter3.txt:a piece of flint, a stone which may be used to start a fire by striking
chapter3.txt:found obsidian, flint, granite, and small stones of basaltic rock, which
chapter9.txt:a flintlock rifle in poor condition. The sandstone mantle held several
```

不要把正则表达式和shell的“文件名匹配模式”（又称为*glob*，文件名通配）混为一谈。在Unix shell中键入**.pm*来匹配所有以*.pm*结尾的文件就是典型的文件名通配。上面的例子使用了*chapter*.txt*这样的文件名通配（你也许已经注意到了，必须用单引号将正则表达式括起来，不然会被shell当成文件名通配）。文件名通配使用了许多与正则表达式相同的字符，但这些字符在使用方式上完全不同[注5]。我们会在第十三章进一步介绍文件名通配，但现在暂且放一下。

使用简单模式

若模式匹配的对象是$_的内容，只要把模式写在一对斜线（/）中就可以了。而模式本身就是一串简单的字符序列：

```
$_ = "yabba dabba doo";
if (/abba/) {
```

注3：当然也有总是匹配或者永远都不匹配的模式。在极少数情况下，这种模式也可能是有用处的，但一般来说都属于使用不当。

注4：除了匹配与否的结果之外，程序还可以取得其他匹配信息，其中一种就是“正则表达式捕获”，稍后我们会在第八章提到。

注5：文件名通配有时也被叫做“模式(pattern)”。不过更糟的是，某些差劲的Unix入门书（可能恰好也是门外汉写的）也称它为正则表达式，其实它们绝对不是。这种说法只会让许多Unix初学者感到迷惑。

```
        print "It matched!\n";
    }
```

表达式/abba/会在$_中寻找这4个字符组成的字符串，如果找到就返回真。这里会找到不止一个字符串，但这并不是关键。只要曾经找到过，匹配结果就为真，否则为假。

由于模式匹配通常用来返回真或假值，所以往往会用在if或while的条件表达式里。其他更多理由留到第八章再说。

所有在双引号引起的字符串中能使用的技巧（尤其是反斜线转义）都可以在模式串里使用。因此，/cake\tsprite/这个模式会匹配coke、一个制表符和sprite这11个字符。

Unicode属性

Unicode字符能够理解自身含义，它们不只是简单的字节序列。每个字符除了字节组合之外，还附带着属性信息。所以除了匹配字符本身以外，我们还能根据字符的属性来达成匹配。

每个属性都有一个名字，完整的清单列在*perluniprops*文档中。若要匹配某项属性，只需要把属性名放入\p{PROPERTY}里面。比如，有许多字符属于空白符（whitespace），相应的属性名为Space，所以要匹配带有这类属性的字符，可以用\p{Space}表示：

```
    if (/\p{Space}/) { # 总共有26个不同的字符带此属性
        print "The string has some whitespace.\n";
    }
```

如果要匹配数字，可以用Digit属性：

```
    if (/\p{Digit}/) { # 总共有411个不同的数字字符
        print "The string has a digit.\n";
    }
```

上面这两个属性所涵盖的字符集合都要比以往见过的多得多。但也有反而更为精简的集合，比如匹配十六进制数字的字符集合[0-9A-Fa-f]，可用下面的属性找出连续两个这样的字符：

```
    if (/\p{Hex}\p{Hex}/) {
        print "The string has a pair of hex digits.\n";
    }
```

我们还能匹配不包含特定属性的字符，只要把小写的*p*改成大写，就表示否定意义，匹配指定属性以外的字符：

```
    if (/\P{Space}/) { # 只要不是空白符都能匹配（自然不计其数！）
        print "The string has one or more non-whitespace characters.\n";
    }
```

关于元字符

当然，如果模式只能匹配简单的直接量字符串，那实际也没太大用处。所以我们引入了特殊字符，称为元字符（*metacharacter*），它们在正则表达式中有着特殊含义。

例如，点号（.）是能匹配任意一个字符的通配符，当然换行符（也就是"\n"）要除外。因此，betty将会被/bet.y/这个模式匹配，而betsy、bet=y、bet.y，或是任何前三个字符为bet、中间接任何一个字符（换行符除外）、后面为y的字符串，也都会被/bet.y/匹配。但是，像bety或betsey就不匹配了，因为在t与y之间没有（或超过）一个字符。点号只能用来匹配一个字符。

因此，如果你想要匹配字符串中的句号，虽然用点号也能匹配到，但这样会额外匹配到不相干的字符（不包括换行符）。要是希望点号仅仅匹配句号本身，只需在前面加上反斜线转义就好了。此规则也适用于任何Perl正则表达式里用到的元字符：在任何元字符前面加上反斜线，就会使它失去元字符的特殊作用。比如，/3\.14159/这个模式里的点号就不是表示通配的元字符。

因此，反斜线是我们的第二个元字符。若要匹配真正的反斜线，请用两个反斜线表示，这个规则也适用于Perl的其他地方：

```
$_ = 'a real \\ backslash';
if (/\\/) {
    print "It matched!\n";
}
```

简单的量词

我们常常需要在某个模式中重复某些东西。而星号（*）正是用来匹配前面的条目零次或多次的。因此/fred\t*barney/能匹配fred和barney之间任意数量的制表符。也就是说，这个模式能匹配出现单个制表符的情况："fred\tbarney"，或者两个制表符的情况："fred\t\tbarney"，或者三个制表符的情况："fred\t\t\tbarney"，甚至一个制表符都没有的情况："fredbarney"。因为星号表示匹配零次或多次，所以字符之间可以有数百个制表符，但除了制表符之外，不能出现其他字符。把星号想成“前面的东西可以重复出现任意多次，也可以是零次”可能会比较容易理解。因为星号是乘法(times)操作符，而英语里“times”也有“次数”的意思[注6]。

如果除了制表符外还想匹配其他字符，该怎么做呢？答案是使用点号。它会匹配任意字

注6：　在正则表达式的数学计算中，这称为克林星（Kleene star）。

符[注7]，因此.*会匹配任意字符零次到无限多次。也就是说，不管fred与barney之间夹着什么东西，都会匹配模式/freb.*barney/。任何一行中只要提到了fred，并且在其后提到barney，就会匹配此模式。我们经常戏称.*为“捡破烂（any old junk）”模式，因为它能匹配字符串中的随便什么东西。

星号应该说是一种量词（*quantifier*），意思是它指定了前一个条目的数量。但它不是唯一的量词，加号（+）也是量词。加号会匹配前一个条目一次以上：/fred+barney/会匹配在fred与barney之间用空格隔开而且只用空格隔开的字符串（空格不是元字符）。它不会匹配fredbarney，因为加号表示在两个名称之间必须有一个以上的空格。把加号想成“算上刚才出现的，再加上（当然不加也可以）任意次重复”或许会比较容易理解。

第三个量词与星号及加号类似，但限制更为严格，它就是问号（?），表示前一个条目是可有可无的。也就是说，它的前一个条目可以出现一次或者不出现。和另外两个量词相同的是，问号也指定了前一个条目出现的次数。不过在使用问号的情况下，它只会出现一次（如果有匹配的条目）或不出现（如果没有匹配的条目），除此之外没有任何其他可能性。所以，/bamm-?bamm/能且只能匹配下面这两种情况：bamm-bamm或bammbamm。这很容易记，可理解为：“前面那一个啊，有吧？没有吧？”

因为这三个量词指定了前一个条目重复出现的次数，所以它们都必须接在某个东西之后。

模式分组

在正则表达式中，圆括号（()，或称小括号）的作用是对字符串分组。因此，圆括号也是元字符。举例来说，模式/fred+/会匹配像freddddddddd这样的字符串，可是这样的字符串实际上不常出现。不过，模式/(fred)+/会匹配像fredfredfred这样的字符串，这可能才是你想要的。那么，模式/(fred)*/又如何呢？它会匹配像hello,world这样的字符串[注8]。

圆括号同时也使得重新使用某些字符串成为可能。我们可以用反向引用（*back reference*）来引用圆括号中的模式所匹配的文字，这个行为我们称为捕获组（*capture*

注7：　除了换行符。接下来不会一直提醒这件事，因为你已经知道。反正字符串里不太会出现换行符，所以一般没什么影响。但别忘了这个细节，因为总有一天，某个换行符会溜进你的字符串里，吃过亏后你一定会记住，点号不能匹配换行符。

注8：　星号代表要匹配重复出现零次以上的fred。在能接受零次的情况下，要使匹配失败是很难的！任何字符串都匹配该模式，即使是空字符串也一样。

group）[注9]。反向引用的写法是在反斜线后面接上数字编号，比如\1、\2这样。相应的数字表示对应顺序的捕获组。

还记得么？使用圆括号包围的点号可以匹配任意非换行字符。我们可以用反向引用\1来再次匹配刚刚在圆括号中匹配的任意字符：

```
$_ = "abba";
if (/(.)\1/) {  # 匹配 'bb'
    print "It matched same character next to itself!\n";
}
```

(.)\1表明需要匹配连续出现的两个同样的字符。(.)首先会匹配a，但是在查看反向引用的时候就会发现下一个字符不是a，导致匹配失败。Perl会跳过这个字符，用(.)来匹配下一个字符b，在查看反向引用的时候会发现下一个字符也是b，这样就达成了成功的匹配。

反向引用不必紧接在对应的捕获组括号后面。下面的模式会匹配y后面的4个连续的非换行符，并用\1反向引用表示匹配d后也出现这4个字符的情况：

```
$_ = "yabba dabba doo";
if (/y(....) d\1/) {
    print "It matched the same after y and d!\n";
}
```

也可以用多个括号来分成多组，每组都可以有自己的反向引用。我们可以用括号定义一个非换行字符的捕获组，后面跟着一个非换行字符的捕获组。然后用反向引用\2和\1来构成有趣的回文模式，比如abba：

```
$_ = "yabba dabba doo";
if (/y(.)(.)\2\1/) { # 匹配 'abba'
    print "It matched after the y!\n";
}
```

讲到这里经常有人会问，该如何区分哪个括号是第几组？幸运的是Larry是用合乎常理的方式来给它们编号的，只要依次点算左括号（包括嵌套括号）的序号就可以了：

```
$_ = "yabba dabba doo";
if (/y((.)(.)\3\2) d\1/) {
    print "It matched!\n";
}
```

注9：　在早期文档或者本书旧版中，你会看到诸如“记忆（memory）”或者“捕获缓存（capture buffer）”之类的说法，但官方正式的讲法是“捕获组（capture group）”。稍后我们还会看到如何构造非捕获组（noncapturing group）。

有时可能得拆开来写，才能看清楚这个模式中各个部分的结构（当然，这种写法的格式不正确 [注10]）：

```
(           # 第一个左括号
    (.)  # 第二个左括号
    (.)  # 第三个左括号
    \3
    \2
)
```

从Perl 5.10开始支持一种新的反向引用写法。不再只是简单的用反斜线和组号，而是用\g{N}这种形式。其中N是想要反向引用的组号。在刚才的例子中使用这种新写法会更加清晰。

想想看，如果反向引用后面跟着的模式的一部分是数字该怎么办。在下面的例子中，我们要用\1来重复刚刚在圆括号中匹配的字符，然后紧接着的必须是字符串直接量11：

```
$_ = "aa11bb";
if (/(.)\111/) {
    print "It matched!\n";
}
```

Perl必须猜测我们的意图：这里到底是\1、\11还是\111呢？在这个问题上Perl的逻辑很简单，它会尽可能创建最多数量的反向引用，所以最终Perl认为这里应该是\111（译注：其实这是很自然的事情，也是必然的选择。如果规则是尽可能少，那么最多只能支持1到9个反向应用，显然这很荒谬。所以不管如何，Perl尽可能往多的原则靠。至于避免歧义，那应该是程序员的责任。）。但因为没有第111（或11）组括号存在，所以Perl在编译阶段就会报错。

而通过使用\g{1}，就能消除反向引用与模式的直接量部分的二义性 [注11]：

```
use 5.010;

$_ = "aa11bb";
if (/(.)\g{1}11/) {
    print "It matched!\n";
}
```

用\g{N}的写法的额外好处是，我们甚至可以用负数。相比指定捕获组的绝对编号，相对反向引用（*relative back reference*）会更加有趣。这样我们可以用-1来做同样的事：

注10：当然其实你可以用/x修饰符来展开复杂的正则表达式，到第八章我们再具体说明怎么做。

注11：通常我们会用\g{1}的更精简形式\g1表示。但此处我们还是建议写上花括号。为避免困惑，我们建议不管何时都统一使用这种完整的写法，除非你已经非常熟悉自如。

```
use 5.010;

$_ = "aa11bb";
if (/(.)\g{-1}11/) {
    print "It matched!\n";
}
```

这样若要在模式中加入更多的内容，就不必总是修改反向引用的编号了。因为要加入另外一个捕获组，就会导致所有绝对编号的反向引用失效，而相对反向引用则不会，因为它使用的是相对于自己的位置，而不是绝对编号，所以维护起来很轻松：

```
use 5.010;

$_ = "xaa11bb";
if (/(.)(.)\g{-1}11/) {
    print "It matched!\n";
}
```

择一匹配

竖线（|）通常可以读成“或”，意思是要么匹配左边的内容，要么匹配右边的内容。也就是说如果左边的模式匹配失败了，还可以用右边的再试试。因此/fred|barney|betty/能匹配任何含有fred或者barney或者betty的字符串。

现在可以使用/fred(|\t)+barney/这样的模式来匹配fred和barney之间出现一次以上空格、制表符或两者混合的字符串。加号表示重复一次或更多。每次只要有重复，(|\t)就可能匹配空格或制表符[注13]。在这两个名字之间至少要有一个空格或制表符。

若要求fred与barney之间的字符必须都一样，你可以把上述模式改成/fred(+|\t+)barney/。如此一来，中间的分隔符就一定得全是空格或全是制表符。

模式/fred(and|or)barney/可用来匹配任何含有fredandbarney或fredorbarney的字符串(注13)。我们还可以用/fredandbarney|fredorbarney/这样的模式来匹配这两个字符串，但这样太长了，执行效率也会降低，具体结果取决于正则表达式引擎内建的优化策略。

字符集

字符集（*character class*），指的是一组可能出现的字符，通过写在方括号（[]）内表

注12：　这类匹配如果改用字符集的形式来做的话，一般效率更高，本章稍后会有说明。

注13：　请注意，在正则表达式里and与or这两个词并不是操作符！因为它们是字符串的一部分，所以在这里用等宽字来表示。

示。它只匹配单个字符，但可以是字符集中列出的任何一个。

比如字符集[abcwxyz]，它可以匹配这7个字符中的任意一个。为方便起见，你可以用连字符（-）表示始末范围，这样之前的字符集也可以改写成[a-cw-z]。当然，这个例子并不会节约多少打字时间，但建立像[a-zA-Z]这样的经典字符集就非常方便了，它可以匹配52个大小写字母中的任何一个。这52个字母并不包括Ä、é、ø、Ü之类的字符，这些都是另外的字母字符，稍后我们会介绍如何匹配他们。

定义字符集时可以使用字符简写，类似双引号内的转义序列，因此字符集合[\000-\177]将会匹配任意一个7位的ASCII字符[注14]。当然，字符集只是完整模式的一部分，在Perl中它从来不会单独出现。比如，你可能会见到下面这样的代码：

```
$_ = "The HAL-9000 requires authorization to continue.";
if (/HAL-[0-9]+/) {
    print "The string mentions some model of HAL computer.\n";
}
```

有时候，指定字符集范围以外的字符会比指定字符集内的字符更容易。可以在字符集开头的地方加上脱字符（caret,^）来表示这些字符除外。也就是说，[^def]会匹配这三个字符以外的任何字符，而[^n\-z]则会匹配n、连字符与z以外的任何字符（请注意，这里的连字符要加上反斜线，因为它在字符集里具有特殊意义。但在/HAL-[0-9]+/里的第一个连字符则不需要反斜线，因为字符集括号外面的连字符没有特殊意义）。

字符集的简写

某些字符集出现的频率非常高，所以我们给它们设定了简写形式。在Perl还是ASCII的时代，我们不必担心字符数量，基本上字符集的简写能表示的无非就是那些字符。但引入Unicode之后，情况就不同了，原来的那些简写覆盖的范围骤增，已经不像原来那么实用了。说起来多少有些难过，我们当中那些用了Perl许多年的人仍然不愿承认这一点。但我们不愿逃避现实，你也不该如此。你看到的其他人写的代码可能是很久以前写的，也可能是昨天写的，仍然在用20世纪90年代的简写形式，却不知道其实这些简写的意义已经发生了很大变化。这在Perl里面绝不是个小问题，常常会引发大家的激烈争论。但不管结论如何，能正确工作的代码才是最重要的。

比如，表示任意一个数字的字符集的简写是\d，那么之前关于HAL的例子中的模式我们可以写成/HAL-\d+/这样的形式：

```
$_ = 'The HAL-9000 requires authorization to continue.';
```

注14： 除非你不用ASCII而是使用EBCDIC字符集。

```
if (/HAL-[\d]+/) {
    say 'The string mentions some model of HAL computer.';
}
```

但除了ASCII里面的0到9之外，还有许多表示数字意义的字符，所以上面这条正则表达式实际上还能匹配HAL-٩等形式的字符串。在Perl5.6之前，\d简写确实严格等同于字符集[0-9]，而今还有许多人习惯用这种字符集作为定义。但现在，它还能匹配比较少见的其他语言中的数字字符，比如٤, ᠒或๓等等，要是你是用阿拉伯文（Arabic）、蒙古文（Mongolian）或者泰文（Thai）计数的话。所以，现在的Perl其实是能用\d匹配这类数字字符的。

在从ASCII到Unicode的转变过程中，如何区分前后不同意义的字符集成了一个需要解决的问题。为此，Perl 5.14引入了一种新的修饰符，当你需要严格按照ASCII的范围来匹配数字字符时，可以选用这种方式。这个修饰符是/a，写在正则表达式末尾（我们稍后会在第八章介绍有关修饰符的内容），表示按照ASCII的语义展开：

```
use 5.014;

$_ = 'The HAL-9000 requires authorization to continue.';

if (/HAL-[\d]+/a) { # 按老的 ASCII 字符语义解释
    say 'The string mentions some model of HAL computer.';
}
```

同样地，\s简写能匹配任意空白符，所以效果上大抵等同于Unicode属性\p{Space}[注15]。在Perl5.6之前，\s仅能匹配以下5个空白符：换页符（form-feed）、水平制表符（tab）、换行符（newline）、回车符（carriage return）以及空格字符本身。所以，明确定义的字符集应该是[\f\t\n\r]。要在新版Perl当中使用严格表示此范围的字符集，可以使用之前\d例子中一样的办法：

```
use 5.014;

if (/\s/a) { # 按老的 ASCII 字符语义解释
    say 'The string matched ASCII whitespace.';
}
```

Perl 5.10还增加了范围更小的空白符集。比如简写\h，它只匹配水平空白符，而\v则只匹配垂直空白符。把\h和\v并起来，就成了\p{Space}：

```
use 5.010;
```

注15：　即使在Unicode语义范畴内，\s仍然不会匹配垂直制表符、下一行(next line)或不间断空格(non breaking space)字符；是不是越来越奇怪了？请参阅*http://www.effectiveperlprogramming.com/blog/991*上的《Know your character classes under different semantics》一文。

```
if (/\h/) {
    say 'The string matched some horizontal whitespace.';
}

if (/\v/) {
    say 'The string matched some vertical whitespace.';
}

if (/[\v\h]/) { # 等同于 \p{Space}，但比 \s 能匹配的要多
    say 'The string matched some whitespace.';
}
```

而Perl 5.10中引入的\R简写能匹配任意一种断行符，也就是说，不管你用的是什么操作系统，\R总会匹配表示断行的那个字符。所以，不管是\r\n还是\n，或者其他Unicode里面表示断行的字符，它都能一概匹配，而不用考虑原本使用的是DOS还是Unix风格的换行符。

简写\w一直被称作“单词（word）”字符，尽管它能匹配的字符并不是严格意义上的单词字符。因为它很受欢迎，所以也会带来一些问题，虽说都不大，但说起来总是怪怪的。以前，“单词”的定义就是指那些可作为标识符（*identifier*）的字符，也就是能用来为Perl变量或者子程序命名的字符集[注16]。在ASCII语义下，\w匹配的是这样一组字符：[a-zA-Z0-9_]，就算这样，人们也常常会在需要只匹配字母的情况下匹配到数字，而那正是现实生活中需要做到的事情。很多时候，人们只是希望匹配数字和字母，但常常忘了其实下划线也属于单词字符，所以也会匹配到。

一个好的正则模式应当仅仅匹配需要的那些字符，不留一点多余。所以只有当我们需要匹配Perl标识符的时候，才是[a-zA-Z0-9_]用得恰如其分的时刻。但我们真的需要经常做这件事吗？

而Unicode对\w的扩展是可以匹配超过100 000个不同形式的单词字符的[注17]。现今的\w定义更为完整准确，但对大多数人来说，很少需要在实际应用中覆盖如此众多的字符。虽然不是很有用，但也不该忽略它，人们依然在日常工作中使用\w的形式，你也会到处看到这种写法。判断这100 000个字符中哪些才是想要匹配的内容，应该是你的职责，但多半都是指[a-zA-Z]这些。我们在第150页上的“单词锚位”一节谈及“单词边界（word boundaries）”时会看到更多相关的例子和说明。

注16：　那些标识符实际上就是C标识符，Perl在早期选用的是同一组字符集。

注17：　所有Unicode的属性名称以及每项属性所囊括的字符数量都列在*perluniprops*文档中。

其实很多情况下，特别是将来写新代码时，应当尽量选用范围明确、可维护性好的模式来定义字符集，避免一味采用简写带来预期之外的结果。

反义简写

有时候你也许只是为了指定以上几种简写以外的字符，也就是类似于[^\d]、[^\w]或是[^\s]这样的模式，来表示一个非数字字符、非单词字符或者非空白符。为此，我们引入了它们的大写版本来表示否定意义，即：\D、\W或者\S。这些大写版本能匹配相应小写版本范围以外的字符。

这些简写既可以作为模式里独立的字符集，也可以作为方括号里字符集的一部分。也就是说，/[\dA-Fa-f]+/可以用来匹配十六进制数字，因为十六进制表示法用的是字母ABCDEF（或abcdef）作为额外数字。

有种比较特别的复合字符集是[\d\D]，表示任何数字或非数字。也就是说，它会匹配任意字符！这是匹配任意字符（包括换行符）的常见做法（而点号则只能匹配换行符以外的所有字符）。此外，还有完全无用的[^\d\D]字符集，它匹配既不是数字也不是非数字的字符。没错，那就是什么都不匹配！

习题

下列习题答案参见第318页上的“第七章习题解答”一节。

记住，对正则表达式的功能感到惊讶是正常的，正因为这样，本章习题才比其他章节的习题更为重要。请做好会遇到意外的心理准备：

1. [10]写个程序，从输入中读取数据，遇到包含fred字符串的行就打印出该行(对于输入中的其他行则不做任何事)。如果输入中的某一行包含字符串Fred、frederick或者Alfred，请问是否会匹配并打印？另外写个文本文件，在里面随意编个“fred flintstone”跟他朋友的故事。然后用这个文件作为输入来测试这一题以及后面几道题。

2. [6]修改上题程序，让它也能接受Fred。那么，当你的输入字符串是Fred、frederick或者Alfred时，是否也匹配？（请将包含这些名称的各行加到刚才的文本文件。）

3. [6]写个程序，只输出其输入中含有点号（.）的每一行，如果没有就忽略。同样以刚才的文本文件进行测试。如果其中有行文字是Mr.Slate，看看会输出吗？

4. [8]写个程序，输出含有要求的单词的行。要求的单词用大写字母开头，但并非全大写。此程序是否会匹配含有Fred的行，而不匹配含有fred或FRED的行？

5. [8]写个程序，打印那些有两个相连且相同的非空格字符的行。应该能匹配包括如Mississippi、Bamm-Bamm或者llama等词的行。

6. [8]附加题：写个程序，输出在输入数据中同时出现wilma以及fred的每一行。

第八章

用正则表达式进行匹配

在第七章里，我们已经大致了解了正则表达式的基本概念。接下来，我们将会看到它是如何融入Perl世界的。

用m//进行匹配

在第七章中，我们用双斜线的写法表示模式，比如/fred/。但事实上，这是m//（pattern match operator，模式匹配操作符）的简写。就像我们在介绍qw//操作符时提到的，可以选择任何成对的定界符。所以，我们可以把它改写为m(fred)、m<fred>、m{fred}、m[fred]，或者也可以用其他不成对定界符来改写成m,fred,、m!fred!、m^fred^等等[注1]。

此处所谓的“简写”是指如果你选择双斜线作为定界符，那么你可以省略开头的m。因为Perl程序员喜欢省略多余字符，所以大部分模式匹配都会写成双斜线的形式，就像/fred/这样。

当然，你应该明智地选择模式中不会出现的字符作为定界符[注2]。在匹配常规网址开头的模式时，可能会用/http:\/\//匹配起始的"http://"。其实可以选择更好的定界

注1：　不成对定界符就是没有左右之分的符号，所以模式两边使用同样的标点符号。

注2：　在使用成对的定界符时，通常不用担心在模式里出现的定界符，因为该定界符在模式里通常是成对出现的。也就是说，m(fred(.*)barney)、m{\w{2,}}与m[wilma[\n\t]+betty]都没问题，即使模式中含有引号也行，因为每一个左定界符都会有一个相应的右定界符。但是尖括号（<与>）并非正则表达式的元字符，所以它们可能不会成对出现。如果模式是m{(\d+)\s*>=?\s*(\d+)}，那么在以尖括号作为定界符的状况下，模式中的大于号前面就需要加上反斜线，才不会过早结束模式。

符，从而提高代码可读性，也能降低维护成本。比如这么写：m%^http://%[注3]。常见的定界符是花括号。如果你用的是程序员专用的文本编辑器，一般都具有从左括号跳到相应的右括号的功能，在维护代码中要快速移动光标时就非常方便。

模式匹配修饰符

Perl有好几个模式匹配修饰符（modifier），有时候也叫做标志（*flag*）[注4]，它们是一些追加在模式表达式末尾定界符后面的字母，用来改变默认的匹配行为。我们之前在第七章中展示过/a，接下来继续介绍其他修饰符。

用/i进行大小写无关的匹配

要实现大小写无关的模式匹配，比如同时匹配FRED、fred和Fred，可以用/i修饰符：

```
print "Would you like to play a game? ";
chomp($_ = <STDIN>);
if (/yes/i) {  # 大小写无关的匹配
    print "In that case, I recommend that you go bowling.\n";
}
```

用/s匹配任意字符

默认情况下，点号（.）无法匹配换行符，这对大多数单行匹配的情况是合适的。但如果字符串中含有换行符，而你希望点号能匹配这些换行符，那么/s修饰符可以完成这个任务。它会将模式中的每个点号[注5]转换成按字符集[\d\D]来处理，就是说会匹配任意字符，包括换行符。当然，你需要有一个包含换行符的字符串，才能看出它的差异：

```
$_ = "I saw Barney\ndown at the bowling alley\nwith Fred\nlast night.\n";
if (/Barney.*Fred/s) {
    print "That string mentions Fred after Barney!\n";
}
```

没有/s修饰符的话，上面的匹配就会失败，因为前后两个名字并不在同一行。

但有时这项特性也会带来一点问题。修饰符/s会把模式中出现的所有.都修改成能匹配任意字符，那么要是我们只想其中几个点号匹配任意字符呢？可以换用字符集[^\n]，不过输入太麻烦，所以Perl 5.12开始引入了\N简写来表示\n的否定意义。

注3：　请记住，斜线并不是元字符，所以使用其他定界符时不需要再加上反斜线转义。

注4：　并且，在Perl 6的领域，这类东西已经有了一个正式的名字：副词（*adverbs*），但这个称呼早在Perl 5的时候就已经开始用起来了。

注5：　若你希望其中部分点号能匹配换行符，只需要将那些部分替换成[\d\D]就行了。

用/x加入空白符

第三个修饰符允许我们在模式里随意加上空白符，从而使它更易阅读、理解：

```
/-?[0-9]+\.?[0-9]*/          # 都挤在一起，很难看清是什么意思
/ -? [0-9]+ \.? [0-9]* /x    # 加入空白后好多了
```

由于加上/x后模式里可以随意插入空白，所以原来表示空格和制表符本身的空白符就失去了意义，Perl会直接忽略。但我们总可以通过转义方式变通实现，比如在空格前面加上反斜线或者使用\t等等。不过最常用的还是\s（或者\s*，抑或\s+），表示匹配空白符。

记住，Perl还会把模式中出现的注释当作空白符直接忽略，所以我们可以在模式中写上注释，帮助阅读者理解作者的意图：

```
/
    -?       # 一个可有可无的减号
    [0-9]+   # 小数点前必须出现一个或多个数字
    \.?      # 一个可有可无的小数点
    [0-9]*   # 小数点后面的数字，有没有都没关系
/x           # 字符串末尾
```

由于井号是注释的标记，所以如果要表示井号字符本身时，就得写成\#，或者使用字符集[#]：

```
/
    [0-9]+   # 小数点前必须出现一个或多个数字
    [#]      # 井号字符本身
/x           # 字符串末尾
```

另外请特别注意，注释部分不要使用定界符，否则会被视为模式终点。比如下面这个例子：

```
/
    -?       # 有减号/没有减号<---糟糕!
    [0-9]+   # 小数点前必须出现一个或多个数字
    \.?      # 一个可有可无的小数点
    [0-9]*   # 小数点后面的数字，有没有都没关系
/x           # 字符串末尾
```

组合选项修饰符

如果需要对单次匹配使用多项修饰符，只需要把它们接在一起写在模式末尾（不用在意先后顺序）：

```
if (/barney.*fred/is) {  # 同时使用 /i 和 /s
    print "That string mentions Fred after Barney!\n";
```

```
}
```

或者像下面这样展开后并带注释：

```
if (m{
    barney # 小伙子 barney
    .*     # 夹在中间的不管什么字符
    fred   # 大嗓门的 fred
}six) {    # 同时使用 /s、/i 和 /x
    print "That string mentions Fred after Barney!\n";
}
```

注意，我们用花括号作为定界符，这样一来，程序员专用编辑器可以根据配对的花括号，快速方便地从正则表达式的开头跳至末尾。

选择一种字符解释方式

Perl 5.14开始增加了一些用于通知Perl如何解释字符意义的修饰符，着重于两个重要方面：对大小写的处理以及对字符集合简写的阐释。本节所有内容仅适用于Perl 5.14和更新版本。

总共有三种字符解释方式：ASCII、Unicode和locale。最后一种（也只有这一种）经常会引发一些问题。修饰符/a告诉Perl采取ASCII方式，而/u则表示采取Unicode方式，最后一种/l表示遵从本地化语言的设定，按照对应的字符集编码作相应处理。如果不提供这类修饰符，Perl会根据*perlre*文档中描述的方式采取最为妥贴的行为。而通过使用修饰符，你可以显式指定程序确切的行为。

首先，我们来谈谈字符集的简写。我们之前已经看到过/a修饰符的用法，它会告诉Perl仅在ASCII范围内阐释\w、\d和\s等字符集简写。修饰符/u会告诉Perl对上面这些简写采取更为宽泛的匹配方式，只要是Unicode范围内定义的单词、数字、空白，都能匹配。修饰符/l告诉Perl按照本地化设定阐释简写的意义，所以任何一个本地设定的字符编码认为是单词的字符，都会被\w简写匹配[注6]。所以，当你使用某项简写并希望按照特定语义对其进行解释的话，可以选择恰当的修饰符作出声明：

```
use 5.014;

/\w+/a        # 仅仅是 A-Z、a-z、0-9、_ 这些字符
/\w+/u        # 任何Unicode当中定义为单词的字符
/\w+/l        # 类同于ASCII的版本，但单词字符的定义取决于本地化设定
              # 所以如果设定为Latin-9的话，Œ也算单词字符
```

注6：　实际上还有一个修饰符/d，它只是告诉 Perl 按照“传统（traditional）”行为行事，即按 Perl自己的推断对字符意义作出最为妥帖的阐释。

哪一种适合你？很抱歉，这个问题我们无法回答，我们不知道你要做什么。这个要看具体时期而定。当然，如果心存疑虑的话，可以始终采用字符集的方式明确声明字符范围。

接下来谈谈比较难的部分。要涉及大小写处理的问题，就势必要知道如何通过大写字母得到对应的小写字母[注7]。如果需要匹配时忽略大小写，Perl必须得知道如何取得对应的小写字母。在ASCII里面，我们知道字母*K*（0x4B）对应的小写字母是*k*（0x6B）。反过来也一样，我们知道ASCII里面小写*k*的大写版本是*K*（0x4B）。看起来好像理所当然，其实不然。

在Unicode里，事情就没那么简单了。不过还不算难，因为我们预先有一张定义好的映射表[注8]。表示热力学温度单位的开尔文符号（Kelvin sign），在Unicode里的字符为*K*（U+212A），同样也有一个对应的小写版本*k*（0x6B）。尽管*K*和*K*看起来一模一样，但对计算机而言，它们的内部编码不同，实际上就是两个完全不同的字符。这就是说，小写到大写的映射并不是一一对应的。一旦拿到小写版本的*k*，就无法取得原来对应的大写版本，上游是两条，该回哪里去呢？不光如此，有些字符，比如合体字*ﬀ*（U+FB00），对应的小写版本有两个字符*ff*。字母ß的小写也是两个小写字母*ss*，但你并不希望匹配这样的小写版本。单个/a修饰符表示按照ASCII方式解释简写意义，如果使用两个/a，则进一步表示仅仅采取ASCII方式的大小写映射处理：

```
/k/aai      # 只匹配ASCII字符K或k，但不匹配开尔文符号
/k/aia      # 其实/a不必相互紧挨，分开写的效果也是一样的
/ss/aai     # 只匹配ASCII的ss、SS、sS、Ss，不匹配ß
/ff/aai     # 只匹配ASCII的ff、FF、fF、Ff，不匹配ﬀ
```

用本地化的话就没这么简单了。你必须得知道正在使用的字符编码是什么。比如内码为0xBC的字符，它究竟是Latin-9里面的Œ还是Latin-1里面的¼，或是其他本地化设定中的字符？如果不知道本地化设定是什么字符编码，就没法进行大小写转换的处理：[214]

```
$_ = <STDIN>;

my $OE = chr( 0xBC ); # 明确取得我们所讲的那个字符

if (/$OE/i) {          # 大小写无关？？？未必。
    print "Found $OE\n";
}
```

注7：　这在Perl里面属于“Unicode bug”，答案其实是在系统内部就写好的。更为详尽的细节不妨参阅*perlunicode*文档。

注8：　参阅*http://unicode.org/Public/UNIDATA/CaseFolding.txt*。

注9：　只要本书在出版过程中不发生意外，你可以从本书电子版直接复制这两个字符，然后看看它们的内码是否一致，虽然看起来完全一样。

在这里的例子里，根据Perl处理$_中字符串以及模式匹配中字符的方式不同，你可能会得到不同的结果。如果程序源代码用UTF-8保存，但输入的字符串是Latin-9编码的，会发生什么？在Latin-9里面，字符*Œ*的序号值为0xBC，而它的小写版本*œ*的序号值为0xBD。在Unicode里面，*Œ*的代码点是U+0152，而*œ*的代码点是U+0153。在Unicode里，U+00BC表示¼，它并没有什么小写版本。如果你在$_中输入的是0xBD并且Perl将正则表达式作为UTF-8字符串来处理的话，是不会得到预想的结果的。不过，你可以添加/l修饰符强制Perl按照本地化设定的规则解析正则表达式的含义：

```
$_ = <STDIN>;

my $OE = chr( 0xBC ); # 明确取得我们所讲的那个字符

if (/$OE/li) {        # 好多了
    print "Found $OE\n";
}
```

如果希望始终按Unicode语义阐释，做法同上面Latin-1的例子一样，在正则表达式末尾明确写上/u修饰符：

```
$_ = <STDIN>;
if (/Œ/ui) {   # 现在用的是Unicode语义
    print "Found Œ\n";
}
```

是不是觉得头很大？没错，谁都不喜欢这么复杂的情形，不过Perl在这里已经尽其所能把该做的事都处理好了。要是能推倒历史重新来过的话，也许就不会闹出这么多麻烦来了。

其他选项

当然还有许多其他修饰符，我们会在用到时再作介绍。你也可以参阅*perlop*文档中有关m//的部分，以及本章随后即将介绍的其他正则表达式相关的操作符[注10]。

锚位

默认情况下，如果给定模式不匹配字符串的开头，就会顺移到下一个字符继续尝试。而通过给定锚位，我们可以让模式仅在字符串指定位置匹配。

注10： 这里我们通过chr()函数构造这个字符，以确保内码同我们讲的一致，避免程序源代码编码不一致导致结果不同的问题。

\A锚位匹配字符串的绝对开头，也就是说，如果开头位置上不匹配，是不会顺移到下一个位置继续尝试的。比如下面这个模式用于判断字符串是否以https开头：

```
m{\Ahttps?://}i
```

对应地，如果要匹配字符串的绝对末尾，可以用\z锚位。比如下面这条模式匹配以.png结尾的字符串：

```
m{\.png\z}i
```

为什么说“字符串的绝对末尾”？我们要强调的是，在\z后面再无任何其他东西。这里面有点历史，另外有个类似的行末锚位\Z，它允许后面出现换行符。这样人们就不必操心去掉单行内容末尾的换行符：

```
while (<STDIN>) {
    print if /\.png\Z/;
}
```

如果严格匹配行末，那么原文中的换行符仍旧会保留下来，所以后续要输出前必须手工去掉换行符：

```
while (<STDIN>) {
    chomp;
    print "$_\n" if /\.png\z/;
}
```

有时候，你会需要同时使用行首锚位和行末锚位，以确保模式能匹配给定字符串的全部。常见的例子是/\A\s*\Z/，它会匹配一个空行。但这个所谓的空行其实是允许包含若干空白符的，就像制表符和空格什么的，虽然看不见，也没有特殊意义，但整体来讲这样的行属于空行。这条模式匹配的就是这样的空行，就像文章段落之间空出的那些行一样，如果不加锚位，则会匹配其他非空行。

\A、\Z和\z都是Perl 5里面的正则表达式特性，但并不是每个人都习惯用它们。在Perl 4里（很多人就是从那时起养成的习惯），所用的表示字符串开头锚位的是脱字符（^）[注11]，用于表示字符串结尾锚位的是$。这两个写法在Per l5里面仍然可用，但已经演化成了行首（beginning-of-line）和行末（end-of-line）锚位。

那么，行首（beginning-of-line）和字符串首（beginning-of-string）的区别何在？出现这

注11： 是的，你之前看到过脱字符在正则模式中的另一种用法。当它出现在字符集定义的开头时，表示对字符集的范围取反。但在字符集定义之外的话，某些时候就是元字符，作为表示字符串首（start-of-string）锚位。

种差别在于，你是怎么看待行的概念，计算机又是怎么看待的。当用$_匹配字符串时，Perl并不关心其中的内容。对Perl来讲，这堆东西不过是一个大字符串罢了，即便里面有好多换行符也是如此。但对人来讲，换行符起到分隔字符单元的作用，看起来就是多行文本：

```
$_ = 'This is a wilma line
barney is on another line
but this ends in fred
and a final dino line';
```

假设给你的任务是找出行末出现fred的字符串，而不是在整个字符串末尾出现fred的字符串。在Perl 5里面，你可以用$锚位和/m修饰符表示对多行内容进行匹配。下面这个模式能成功匹配，因为上面这个多行字符串中fred位于行末位置：

```
/fred$/m
```

此外，/m修饰符还改变了原先在Perl 4中锚位的工作方式。上面的模式会匹配给定字符串中所有fred以及随后出现的换行符，或者是字符串绝对末尾位置上的fred。

修饰符/m同样会改变^锚位的行为，也就是说，该锚位会同时匹配字符串绝对开头位置和换行符之后的位置。所以下面这个模式会匹配成功，因为多行文本中barney出现在了行首：

```
/^barney/m
```

如果没有/m，^和$的行为就如同\A和\z一样。并且，如果模式写好后哪天又追加了/m开关的话，就会改变原来的意图，所以尽可能严谨地使用恰到好处的锚位，不留一点多余，总归要安全些。但正如我们之前提到的，很多人从Perl 4开始就养成了习惯，所以身边仍然会看到许多使用^和$锚位的例子，但它们的本意其实只是\A和\z罢了。本书后半部分，我们将选用\A和\z严谨行事，除非真的需要用到匹配多行文本的情况。

单词锚位

锚位并不局限于字符串首尾。比如\b是单词边界锚位，它匹配任何单词的首尾[注12]。因此，/\bfred\b/可匹配fred，但无法匹配frederick、alfred或manfredmann。这和文字处理器的搜索命令类似，常称为“整词匹配”。

不过，这里所说的“单词”并不完全等同于一般的英文单词，它是由一组\w字符构成的字符集，也就是由英文字母、数字与下划线组成的字符串。\b锚位匹配的是一组连续的

注12：　有些正则表达式引擎会以一个锚位来表示单词开头，另一个锚位来表示单词结尾，但Perl不加区分，首尾都用\b表示。

\w字符的开头或结尾。有关\w简写的准确定义，请复习一下本章之前的介绍。

在图8-1中，每个词下方会出现灰色下划线，\b能匹配的位置以箭头标识。因为每个单词都会有开头与结尾，所以字符串中的单词边界一定是偶数个。

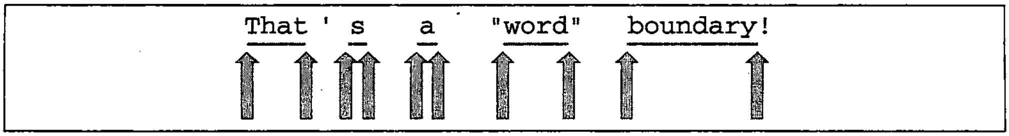


图8-1：用\b匹配单词边界

此处所谓的“单词”，是指一连串英文字母、数字与下划线的组合，也就是匹配/\w+/模式的字符串。该句共有5个单词：That、s、a、word以及boundary [注13]。要注意的是，word两边的引号并不会改变单词边界。这些单词是由一组\w字符构成的。

因为单词边界锚位\b只匹配每组\w字符的开头或结尾，所以每个箭头会指向灰色下划线的开头或结尾。

单词边界锚位非常有用，它保证我们不会意外地在delicatessen中找到cat，在boondoggle中找到dog，或在selfishness中找到fish。有时候，你只会用到一个单词边界锚位，像用/\bhunt/来匹配hunt、hunting或hunter，而排除了shunt；或是用/stone\b/来匹配standstone或flintstone，但不包括capstones。

非单词边界锚位是\B，它能匹配所有\b不能匹配的位置。因此，模式/\bsearch\B/会匹配searches、searching与searched，但不匹配search或researching。

绑定操作符=~

默认情况下模式匹配的操作对象是$_，绑定操作符（*binding operator*，=~)告诉Perl，拿右边的模式来匹配左边的字符串，而不是匹配$_ [注14]。例如：

```
my $some_other = "I dream of betty rubble.";
if ($some_other =~ /\brub/) {
    print "Aye, there's the rub.\n";
}
```

注13：　从这里可以看出来，为什么我们想改变单词的定义了，因为 That's 应该算成一个单词，而不该拆成两个单词与所有格符号。就算在最严肃的英文资料中，也不难看到这种缩写风格。

注14：　除了模式匹配，其他某些运算也会用到绑定操作符，我们稍后会提到。

绑定操作符虽然看起来像某种赋值操作符，其实并非如此！它只是说明本来这个模式匹配会针对$_变量，但现在请匹配左边给出的字符串。若没有绑定操作符，表达式就会使用默认的$_。

在下面这个（有点不寻常的）例子里，$likes_perl会被赋予一个布尔值，这个结果取决于用户键入的内容。这段代码属于快速消费型（quick-and-dirty），因为判断之后就丢弃了用户的输入。这段代码的大致功能是读取输入行，匹配字符串与模式，然后舍弃输入行的内容[注15]。这里的匹配操作完全没有用到默认变量$_，当然也没有改动它的值：

```
print "Do you like Perl? ";
my $likes_perl = (<STDIN> =~ /\byes\b/i);
……  # 时光流逝……
if ($likes_perl) {
    print "You said earlier that you like Perl, so...\n";
    ...
}
```

因为绑定操作符的优先级相当高，也就没必要用圆括号来括住模式测试表达式。所以下面这一行如同上面的表达式一样，会将匹配结果（而非该行输入的内容）存进变量：

```
my $likes_perl = <STDIN> =~ /\byes\b/i;
```

模式中的内插

正则表达式内部可以进行双引号形式的内插，这样我们就可以很快写出下面这样类似*grep*命令的程序：

```
#!/usr/bin/perl -w
my $what = "larry";

while (<>) {
    if (/\A($what)/) {  # 模式的锚位被定在字符串的开头
        print "We saw $what in beginning of $_";
    }
}
```

不管$what的内容是什么，当我们进行模式匹配时，该模式都会由$what的值决定。在这里，它等效于/\A(larry)/，也就是在每行的开头寻找larry。

但$what的内容未必一定要事先写在直接量中，我们可以通过@ARGV取得命令行参数：

注15：请记住，除非while循环的条件表达式中只有整行输入操作符（<STDIN>），否则输入行不会被Perl自动存入$_。

```
my $what = shift @ARGV;
```

如果第一个命令行参数是fred|barney，则模式会变成/\A(fred|barney)/，也就是在每一行开头寻找fred或barney [注16] 。上面搜寻larry时两边看似多余的圆括号其实非常重要。如果省略圆括号，当输入的模式改成现在这样两者选一的话，就会产生另外的效果：要么在字符串开头匹配fred，要么在字符串的任何地方匹配barney。

将程序代码改成从@ARGV读取匹配模式后，这个程序就和Unix的*grep*命令很像了。但我们必须注意字符串里的元字符。如果$what的值为'fred(barney'，该模式就会变成/\A(fred(barney)/，你也知道这样是无法正常工作的，它会报告非法正则表达式的错误并中断程序运行。不过运用某些高级技术的话 [注17] ，是可以捕获这类错误的（或者从一开始就解除元字符的魔法），以避免程序崩溃。不过目前你只需要记住，一旦赋予用户使用正则表达式的权力，他们就该负起正确使用的责任。

捕获变量

圆括号出现的地方一般都会触发正则表达式引擎捕获匹配到的字符串。捕获组会把匹配圆括号中模式的字符串保存到相应的地方。如果不止一个括号，也就不止一个捕获组。每个捕获组包含的都是原始字符串中的内容，而不是模式本身。我们可以通过反向引用取得这些捕获内容，但也可以在匹配操作结束后立即通过相应的捕获变量取得这些内容。

既然这些捕获变量保存的是字符串，那就说明它是标量变量。在Perl里面，它们的名称就是$1和$2这样的形式。模式中有多少个捕获括号，就有多少个对应名称的捕获变量可用。所以，变量$4的意思就是模式中第4对括号所匹配的字符串内容，这个内容和模式运行期间反向引用\4所表示的内容是一样的。但它们并非同一个事物的两种名称：\4反向引用的是模式匹配期间得到的结果，而$4则是模式匹配结束后对得到的捕获内容的索引。更多关于反向引用的信息请参阅*perlre*文档。

可以说，捕获变量是正则表达式无比强大的重要原因之一，因为有了它，我们才得以拥有提取字符串中某些特定部分的能力：

```
$_ = "Hello there, neighbor";
if (/\s(\[a-zA-Z]+),/) {        # 捕获空白符和逗号之间的单词
```

注16： 明眼的读者将会知道，你通常无法在命令行上键入fred|barney这个参数，因为竖线符号是shell的元字符。请参考手头的shell说明文档，学习如何为命令行参数加上引号。

注17： 在此状况下，你可以使用eval块来捕获这个错误。你也可以用quotemeta（或是它的等效形式\Q）为所内插的文字加上引号，这样一来，特殊意义的元字符就不会捣乱了。

```
    print "the word was $1\n";  # 打印 the word was there
}
```

或者也可以一次捕获多个字符串：

```
$_ = "Hello there, neighbor";
if (/(\S+) (\S+), (\S+)/) {
    print "words were $1 $2 $3\n";
}
```

运行结果是words where Hello there neighbor。请注意，在输出结果里没有逗号。因为模式里的逗号放在圆括号外面，所以第二个捕获中不会有逗号。使用这个技巧，我们可以精确筛选捕获的（和跳过的）数据。

有时得到的捕获变量可能是空的 [注18]，因为给出的字符串完全有可能不符合模式要求。所以，捕获变量有可能出现包含空字符串的情况：

```
my $dino = "I fear that I'll be extinct after 1000 years.";
if ($dino =~ /([0-9]*) years/) {
    print "That said '$1' years.\n";  # $1 为 1000
}

$dino = "I fear that I'll be extinct after a few million years.";
if ($dino =~ /([0-9]*) years/) {
    print "That said '$1' years.\n";  # $1 为空字符串
}
```

捕获变量的存续期

这些捕获变量通常能存活到下次成功匹配为止 [注19]。也就是说，失败的匹配不会改动上次成功匹配时捕获的内容，而成功的匹配会将它们的值重置。这就意味着，捕获变量只应该在匹配成功时使用，否则就会得到之前一次模式匹配的捕获内容。下面的（失败）案例本来应该输出从$_捕获的某个单词。但是如果匹配失败，它会输出之前遗留在$1里的字符串：

```
my $wilma = '123';
$wilma =~ /([0-9]+)/;     # 匹配成功，$1的内容是123
$wilma =~ /([a-zA-Z]+)/;  # 错了！这里没有判断匹配结果是否成功
```

注18：　空字符串并不等同于未定义字符串。若模式中有三个或者更少的圆括号，那么$4的值就是undef。

注19：　实际存续范围的规则更为复杂（具体请参考说明文档），不过一般来说，模式匹配后的若干行内就会用到这些变量，所以大体不会成为问题。

```
print "Wilma's word was $1... or was it?\n";  # 所以捕获变量$1里的内容仍旧是123!
```

这就是为什么模式匹配总是出现在if或while条件表达式里的原因：

```
if ($wilma =~ /([a-zA-Z]+)/) {
    print "Wilma's word was $1.\n";
} else {
    print "Wilma doesn't have a word.\n";
}
```

既然这些捕获内容不会永远留存，那么$1之类的捕获变量只应该在模式匹配后的数行内使用。如果程序维护员在原先的正则表达式和$1的使用之间加入了一个新的正则表达式，$1将会是第二次匹配捕获的值，而非第一次。因此，如果需要在数行之外使用捕获变量，通常最好的做法是将它复制到某个普通变量里。这其实也能改善程序代码的可读性：

```
if ($wilma =~ /([a-zA-Z]+)/) {
    my $wilma_word = $1;
    ...
}
```

稍后，在第九章我们会学习如何在模式匹配发生的同时，直接将捕获的内容存到变量，免于使用$1这种粗放的写法。

不捕获模式

目前所见的圆括号都会捕获部分的匹配字符串到捕获变量中，但有时候却需要关闭这个功能，而仅仅只是用它来进行分组。比如某个模式中有些部分是可选的，之后的部分却是要捕获的。好比在某些时候巨无霸（bronto）是可选的，而之后的部分，比如牛排（steak）或者汉堡（burger）却正是我们感兴趣的一样：

```
if (/(bronto)?saurus (steak|burger)/) {
    print "Fred wants a $2\n";
}
```

尽管有时“bronto”不存在，还是得把$1变量留给它。Perl只是按左圆括号的序号来决定捕获变量名，这导致我们真正想捕获的内容只能进入$2。在更复杂的模式中，这种情况非常让人困惑。

还好Perl的正则表达式允许使用圆括号分组但不进行捕获。我们把这叫做不捕获圆括号

（*noncapturing parentheses*），书写的时候也有些差别，需要在左括号后面加上问号和冒号（?:）[注20]，告诉Perl这一对圆括号完全是为了分组而存在的。

可以在这里使用不捕获圆括号来跳过“bronto”，这样就可以用$1捕获实际需要的内容：

```
if (/(?:bronto)?saurus (steak|burger)/) {
    print "Fred wants a $1\n";
}
```

之后若要修改正则表达式，以支持巨无霸汉堡的熏肉特色版，就可以在这个模式中加入不捕获选项。这时候你感兴趣的字符串仍然会进入捕获变量$1。若是没有这个功能，就必须在每次加入分组括号之后修改捕获变量名：

```
if (/(?:bronto)?saurus (?:BBQ )?(steak|burger)/) {
    print "Fred wants a $1\n";
}
```

Perl的正则表达式有许多其他用在圆括号内部的修饰符，能达成更复杂的功能，比如前瞻、后顾、内嵌注释，甚至可以在模式中运行代码。详细信息可以参考*perlre*文档。

命名捕获

我们可以利用圆括号的捕获功能提取特定字符串并保存到诸如$1、$2这样的变量中。不过就算是较为简单的模式，要维护数字变量和圆括号之间的对应关系也是一件比较困难的事。比如下面这个正则表达式，匹配$names中出现的两个名字：

```
use 5.010;

my $names = 'Fred or Barney';
if ( $names =~ m/(\w+) and (\w+)/ ) { # 不会匹配成功
    say "I saw $1 and $2";
}
```

实际上看不到say的输出，因为模式字符串中期望的是and，而实际变量中给出的是or。为了应对这种情况，我们觉得应该允许两者并存，所以在正则表达式中加入择一匹配（alternation），不管and还是or都没关系。当然，作为择一匹配的部分必须要加上圆括号以表示候选列表范围：

```
use 5.010;

my $names = 'Fred or Barney';
```

注20：　这是?在正则表达式中的第四种用法：表示问号字符本身（需要转义）、表示数量可有可无、表示非贪婪匹配（第九章中介绍）和这里的表示放弃捕获。

```
if ( $names =~ m/(\w+) (and|or) (\w+)/ ) { # 现在能匹配了
    say "I saw $1 and $2";
}
```

呀！虽然现在能看到输出的消息，但第二个名字不对，因为我们刚又加上了一对圆括号捕获其中内容。现在从$2拿到的是择一匹配部分中的内容，而原本希望拿到的第二个名字推后存放到了$3变量中，而我们事后并未输出该变量：

```
I saw Fred and or
```

当然我们可以用不捕获圆括号的写法来解决，但换汤不换药，问题是我们仍旧要记住括号的序号。要是模式中需要捕获的内容有很多该怎么办呢？

为了避免记忆$1之类的数字变量，Perl 5.10增加了对捕获内容直接命名的写法。最终捕获到的内容会保存在特殊哈希%+里面：其中的键就是在捕获时用的特殊标签，对应的值则是被捕获的字符串。具体的写法是(?<LABEL>PATTERN)，其中LABEL可以自行命名[注21]。下面把第一个捕获标签定为name1，第二个标签定为name2。所以，提取捕获内容时需要访问的变量变成了$+{name1}和$+{name2}：

```
use 5.010;

my $names = 'Fred or Barney';
if ( $names =~ m/(?<name1>\w+) (?:and|or) (?<name2>\w+)/ ) {
    say "I saw $+{name1} and $+{name2}";
}
```

现在可以看到正确结果了：

```
I saw Fred and Barney
```

一旦使用了捕获标签，就可以随意移动位置并加入更多的捕获圆括号，不会因为次序变化导致麻烦：

```
use 5.010;

my $names = 'Fred or Barney';
if ( $names =~ m/((?<name2>\w+) (and|or) (?<name1>\w+))/ ) {
    say "I saw $+{name1} and $+{name2}";
}
```

在使用捕获标签后，反向引用的用法也随之有所变化。之前我们用\1或者\g{1}这样的写法，现在我们可以使用\g{label}这样的写法：

注21：　Perl也允许用Python的语法（?P<LABEL>...）完成同样的功能。

```
use 5.010;

my $names = 'Fred Flintstone and Wilma Flintstone';

if ( $names =~ m/(?<last_name>\w+) and \w+ \g{last_name}/ ) {
    say "I saw $+{last_name}";
}
```

我们也可以用另一种语法来表示反向引用。\k<label>等效于\g{label}[注22]：

```
use 5.010;

my $names = 'Fred Flintstone and Wilma Flintstone';

if ( $names =~ m/(?<last_name>\w+) and \w+ \k<last_name>/ ) {
    say "I saw $+{last_name}";
}
```

自动捕获变量

有三个免费的捕获变量[注23]，就算不加捕获圆括号也能使用。听起来不错，不过它们的名字不太好记，甚至有点诡异。

虽然Larry可能不会反对给它们取比较正常的名字，像$gazoo或$ozmidiar，但这些都是在你自己的程序里可能会用到的名称。为了让普通的Perl程序员在为第一个程序的第一个变量命名时不必绕开Perl所有的特殊变量名[注24]，Larry给内置变量起了一些稀奇古怪的名字，也可以说是“惊世骇俗”的名字。在这里，他选用的是标点符号：$&、$`和$'。它们看起来很奇怪，也很丑陋，非常诡异，不过这就是它们的名字[注25]。字符串里实际匹配模式的部分会被自动存进$&里：

```
if ("Hello there, neighbor" =~ /\s(\w+),/) {
```

注22：实际上，\k<label>与\g{label}有着细微差别。在两个以上的组有同名标签时，k<label>和\g{label}总是会指代最左边的那组，但\g{N}就可以实现相对反向引用。另外，如果你是Python的爱好者，也可以使用(?P=label)这样的语法。

注23：是的，你是对的。实际上并没有什么免费的捕获，“免费”的意思不过是说，某些情况下可以省略圆括号罢了，但它本身还是需要成本的。不用担心，我们稍后就会介绍它幕后的实际开销。

注24：你仍然需要避开少数几个传统的变量名（如$ARGV），不过它们为数不多，也都是全大写的。所有Perl的内置变量在*perlvar*文档里都有介绍。

注25：若你真的无法忍受这些奇怪的名称，可以试试English模块，它会为Perl所有的奇怪变量赋予接近正常的名称。但使用这个模块的人一直不多；相反，Perl 程序员已经逐渐爱上了这些由标点符号组成的变量名，无论它们看起来有多么奇怪。

```
    print "That actually matched '$&'.\n";
}
```

当上面的程序运行时，会告诉我们字符串里匹配的部分是"there,"（一个空格、一个单词以及一个逗号）。第一个捕获内容存在$1中，是具有5个字母的单词there，但$&里保存的是整个匹配区段。

匹配区段之前的内容会存到$`里，而匹配区段之后的内容则会存到$'里。另外一个理解方法是，$`保存了正则表达式引擎在找到匹配区段之前略过的部分，而$'则保存了字符串中剩下的从未被匹配到的部分。如果将这三个字符串依次连接起来，就一定会得到原来的字符串：

```
if ("Hello there, neighbor" =~ /\s(\w+),/) {
    print "That was ($`)($&)($').\n";
}
```

程序运行时会把字符串显示为(Hello)(there,)(neighbor)，展现出这三个自动捕获变量的实际用途。

这三个自动捕获变量也都有可能是空字符串，它们的存续范围和编号的捕获变量完全一样。一般而言，它们的值会一直持续到下一次模式匹配成功之前。

我们之前提过这三个变量可以免费使用。不过，免费也是有代价的。在这里的代价是：一旦在程序中任何部分使用了某个自动捕获变量，其他正则表达式的运行速度也会变慢[注26]。

这虽不会严重拖慢速度，但足以成为隐患，许多Perl程序员干脆永远不碰这些自动捕获变量[注27]。他们会找出变通办法。比如你可以将整个模式加上括号，然后以$1来代替$&（当然，你可能需要调整模式的捕获编号）。

如果你用的是Perl 5.10或以上的版本，那么就更方便了。修饰符/p只会针对特定的正则表达式开启类似的自动捕获变量，但它们的名字不再是$`，$&和$'，而是用${^PREMATCH}、${^MATCH}和${^POSTMATCH}表示。于是，之前的例子可以改写成：

注26：其实是在每个块的进入点与离开点之间，所以差不多就是每个地方了。

注27：虽然大部分人没有真的测过他们的程序，看看改进过的代码是否真的会节省大量时间。对他们而言，这些变量像是有毒似的。但我们不能责怪他们不做性能测试，许多利用这三个变量的程序一周只会多花费几分钟的CPU时间，因此测试速度与进行优化反而会浪费更多的时间。既然如此，为什么要害怕可能额外多出的几毫秒呢？

```
use 5.010;
if ("Hello there, neighbor" =~ /\s(\w+),/p) {
    print "That actually matched '${^MATCH}'.\n";
}

if ("Hello there, neighbor" =~ /\s(\w+),/p) {
    print "That was (${^PREMATCH})(${^MATCH})(${^POSTMATCH}).\n";
}
```

这些变量名看起来有点古怪，名字前面加上了^，又在外面围上了花括号。随着Perl的进化，用于特殊事物的名字已经不敷使用，加上开头的^的目的是避免名称冲突（实际上，程序员能够自由命名的变量是不能以^开头的，它是一个非法字符），外面围住花括号的目的是表示其中的内容是完整的变量名。

捕获变量（包括这里介绍的自动捕获变量以及之前介绍的有编号的那些）常常用于正则表达式的替换操作，有关这一点，我们可以在第九章进一步学习。

通用量词

模式中的量词（*quantifier*）代表前面条目的重复次数。到目前为止，我们已经见过三个量词：*、+以及?。如果这三个量词都不符合需要，你还可以使用花括号（{}）的形式指定具体的重复次数范围。

因此，模式/a{5,15}/可匹配重复出现5到15次的字母a。如果a连续出现3次，则不匹配，因为次数太少了；若a出现5次，就会匹配成功；出现10次时，也会匹配成功。因为15是上限，所以如果a出现20次，就只有前15个会匹配。

如果省略第二个数字但保留逗号，则表示匹配次数没有上限。所以，/(fred){3,}/这个模式会匹配重复出现3次以上的fred（在每个fred之间不能有空格等额外字符）。因为没有上限，所以即使字符串里有88个fred，也会匹配成功。

如果同时省略逗号与上限次数，那么花括号里的数字就表示一个固定的次数：/\w{8}/会匹配正好有8个单词字符的字符串（或许是大型字符串里的一部分），而/,{5}chameleon/所匹配的就是“comma comma comma comma commma chameleon”。对歌手George来说，这是一句很好的歌词。（译注：此处的“comma”所取的是“karma”的谐音。“Karma Chameleon”是歌手Boy George在1983年连续6周的排行榜冠军单曲，其中有一句歌词就是“Karma Karma Karma Karma Karma Chaneleon”。）

事实上，我们之前提到的三个量词字符都只是常用的简写而已。星号和量词{0,}相同，表示零次或多次。加号相当于{1,}，表示一次以上。然后，问号也可以写成{0,1}。因为这三个简写几乎能满足绝大多数的需求，所以平时不太会需要花括号量词。

优先级

看完正则表达式中这一大堆的元字符，你可能会觉得需要一张记分卡帮助整理思路。确实有一张优先级表，它告诉我们模式中哪些部分的“紧密度”最高。不像操作符的优先级表，正则表达式的优先级表相当简单，只有4个级别。我们这里顺便复习一下Perl模式中使用的元字符。对于表8-1所展示的优先级顺序，大致阐释如下：

1. 最高等级是圆括号（“()”），用于分组和捕获。圆括号里的东西总是比其他东西更富有紧密性。

2. 第二级是量词，也就是重复操作符：星号（*）、加号（+）、问号（?），以及使用花括号表示的量词，比如{5,15}、{3,}和{5}。它们都会和它前面的条目紧密相连。

3. 第三级是锚位和序列。我们已经介绍过的锚位有：\A、\Z、\z、^、$、\b和\B[注28]。序列（彼此相邻的条目）事实上也是操作符，就算没有使用元字符也是如此。这就是说，单词里字母之间的紧密程度和锚位与字母之间的紧密程度是相同的。

4. 第四级是择一竖线（|）。由于它位于优先级表底部，所以从效果上来看，它会把各种模式拆分成数个组件。另外，择一竖线的优先级之所以放在锚位与序列的下面，是因为我们希望类似/fred|barney/的模式中，单词里的字母间的紧密程度高于择一竖线。否则，该模式的解释方式就成了“匹配fre，后面所跟的字母必须是d或者b，然后再跟arney”。所以，择一竖线位于优先级表的底部，这样单词里的字母才会紧密连接在一起，成为一个整体。

5. 最低级别的称为原子（*atoms*）。正是由这些原子构成了大多数基本的模式，比如单独的字符、字符集以及反引用等等。

表8-1 正则表达式优先级

正则表达式特性	示例
圆括号（分组或捕获）	(…)，(?:…)，(?<LABEL>…)
量词	a*，a+，a?，a{n,m}
锚位和序列	abc，^，$，\A，\b，\z，\Z
择一竖线	a|b|c
原子	a，[abc]，\d，\1，\g{2}

注28： 还有一个\G锚位尚未介绍。

优先级范例

当你需要解读相当复杂的正则表达式时，你就得照着Perl的方式，使用优先级表按部就班地进行分析。

举例来说，/\Afred|barney\z/大概不会是程序员想要的模式。因为择一竖线的优先级比较低，这样整个模式就会被拆成两半。这个模式要么匹配字符串开头的fred，要么匹配字符串结尾的barney。程序员实际想要的多半是/\A(fred|barney)\z/，也就是匹配只包含fred或只包含barney的每一行[注29]。那么，模式/(wilma|pebbles?)/该怎么理解呢？实际上，那个问号量词仅仅对接在前面的字符起作用[注30]，所以这个模式可匹配到willma、pebbles以及pebble这三个字符串，或是长字符串中的一小部分（因为模式中没有锚位）。

模式/\A(\w+)\s+(\w+)\z/可用来匹配开始是一个单词，再来是一些空白，然后又是一个单词（前面或后面没有其他东西）的行。举例来说，它大概就是用来匹配fred flintstone之类的字符串。这里的圆括号并不是为了分组而存在，可能只是为了把匹配的字符串捕获下来。

在尝试理解一个很复杂的模式时，试着加上一些圆括号会对弄清优先级有好处。但请记住，圆括号同时也会有捕获的效果。因此建议尽可能用非捕获圆括号来分组。

还有更多

虽然我们介绍了所有日常编程中会用到的正则表达式特性，但实际远远不止这些。有些特性我们放到《Intermediate Perl》一书介绍。不过你可以翻阅*perlre*、*perlrequick*以及*perlretut*文档，看看Perl在模式匹配方面还能做哪些事情[注31]。

模式测试程序

编写Perl程序的时候，每个程序员都免不了要使用正则表达式，但有时候很难轻易看出一个模式能做什么事。而且常常会发现，模式匹配的范围总比预期的大些或小些。要不就是开始匹配的位置早些或晚些，要不就是根本无法匹配。

注29： 并且，也许字符串结尾还有一个换行符，就像之前介绍\z锚位时提到的。

注30： 因为在pebbles里，问号量词与字母s间的连接要比s与其他字母的连接来得紧密。

注31： 并且看看CPAN上的YAPE::Regex::Explain模块，这是一个用英语解释正则表达式含义的工具。

下面这个程序非常实用，可用于检测某些字符串是否能被指定模式匹配以及在什么位置上匹配[注32]：

```
#!/usr/bin/perl
while (<>) {                        # 每次读一行输入
    chomp;
    if (/YOUR_PATTERN_GOES_HERE/) {
        print "Matched: |$`<$&>$'|\n";  # 特殊捕获变量
    } else {
        print "No match: |$_|\n";
    }
}
```

这个模式测试工具是给程序员使用的，而不是给最终用户，你一看便知分晓，因为并没有提示符，也没有用法说明。它会把所有输入一行行读进来，然后以你在YOUR_PATTERN_GOES_HERE指定的模式进行匹配。如果该行匹配模式，就会利用三个特殊的捕获变量（$`、$&和$'）展示实际的匹配结果[注33]。假设你使用的模式是/match/，而输入的字符串是beforematchafter，那么你会看到的程序输出就是|before<match>after|，尖括号里面的内容是字符串匹配模式的部分。如果结果跟你预想的不一样，马上就可以看出来。

习题

以下习题答案参见第320页上的“第八章习题解答”一节。

有几道题需要用到本章给出的模式测试程序。你当然可以自行输入该程序源代码，但要小心不要打错标点符号。当然你也可以上网，访问我们的网站*http://www.learning-perl.com*自行下载[注34]。

1. [8]利用模式测试程序，写个模式，使它能匹配match这个字符串。你可以测试一下字符串beforematchafter，看是否会正确显示匹配到的部分以及前后的部分？
2. [7]利用模式测试程序，写个模式，使其能够匹配任何以字母a结尾的单词（以\w组

注32：如果你正在阅读的不是电子书，无法复制粘贴的话，请访问本书网站*http://www.learning-perl.com*，到下载区自行下载这段程序的源代码。

注33：测试正则表达式是否正常如期工作的时候，并不需要关心性能问题。不管你用的是Perl 5.10之前的版本还是之后的版本，我们都希望这段程序能正常工作，所以捕获变量也是按照以往老式写法表示。

注34：如果你真的要自己键入该程序源代码，请记住，反引号字符（`）和单引号字符（'）是完全不一样的。在全尺寸键盘上（至少在美式键盘布局上），反引号键位于数字1键的左边。

成的单词）。此模式是否能够匹配到wilma？是否无法匹配到barney？此模式是否能够匹配到Mrs.WilmaFlintsone？还有wilma&fred呢？把第七章里的样本文本文件拿到这里测测看（并把这些测试字符串加到该文本文件里）。

3. [5]修改上题程序，使其在匹配到以a结尾的单词的同时也将其存储在$1里。接着修改程序的输出，让变量的内容出现在单引号中，例如：$1 contains 'Wilma'。

4. [5]修改上题程序，使用命名捕获而不是$1这样的老办法。接着修改程序输出，让标签名字出现在结果中，例如：'word' contains' Wilma'。

5. [5]附加题：修改上一题的程序，使其在定位到以a结尾的单词后，最多再将之后的5个字符（如果有那么多的话）捕获至一个独立的内存变量。修改程序输出，把所用到的这两个内存变量都输出来。假设你输入的字符串是I saw Wilma yesterday，那么后面取到的5个字符就是“yest”。如果你输入的是I,Wilma!，那么第二个内存变量的内容只会有一个字符。看看你的模式是否还可以成功匹配wilma这个简单的字符串？

6. [5]写个新程序（不是之前给出的测试程序），输出其输入中以空白结尾的行（换行符不算）。输出的时候，在行尾多加一些记号，这样比较容易看出空白符。

第九章

用正则表达式处理文本

正则表达式也可用于修改文本。之前我们只是介绍如何利用正则表达式进行模式匹配，现在我们来看如何利用正则表达式的模式定位部分字符串并做相应修改。

用s///进行替换

如果把m//模式匹配（pattern match）想象成文字处理器的“查找”功能，那么s///替换（substitution）操作符就是“查找并替换”功能。此操作符只是把存在变量[注1]中匹配模式的那部分内容替换成另一个字符串：

```
$_ = "He's out bowling with Barney tonight.";
s/Barney/Fred/;  # 把 Barney 替换为 Fred
print "$_\n";
```

如果匹配失败，则什么事都不会发生，变量也不受影响：

```
# 接着之前的代码；$_现在为 "He's out bowling with Fred tonight."
s/Wilma/Betty/;  # 试图用Betty替换Wilma（将会失败）
```

当然，模式字符串与替换字符串还可以更加复杂。下面的替换字符串用到了第一个捕获变量，也就是$1，模式匹配时会对它赋值：

```
s/with (\w+)/against $1's team/;
print "$_\n";  # 打印 "He's out bowling against Fred's team tonight."
```

注1：　不像m//可以匹配任何字符串表达式，s///修改的数据必须是预先存放在某个变量中的。这个变量一般位于该操作符的左边，所以也称为左值（lvalue）。一般来说，左值都是某个变量，但其实不管是什么，只要能放在赋值语句左边的东西也都可以。

这里还有一些替换操作的例子，不过仅仅出于示范的目的，实际使用时一般不会出现这么多互不相关的替换操作：

```
$_ = "green scaly dinosaur";
s/(\w+) (\w+)/$2, $1/;  # 替换后为"scaly, green dinosaur"
s/^/huge, /;            # 替换后为"huge, scaly, green dinosaur"
s/,.*een//;             # 空替换：此时为"huge dinosaur"
s/green/red/;           # 匹配失败：仍为"huge dinosaur"
s/\w+$/($`!)$&/;        # 替换后为"huge (huge !)dinosaur"
s/\s+(!\W+)/$1 /;       # 替换后为"huge (huge!) dinosaur"
s/huge/gigantic/;       # 替换后为"gigantic (huge!) dinosaur"
```

s///返回的是布尔值，替换成功时为真，否则为假：

```
$_ = "fred flintstone";
if (s/fred/wilma/) {
    print "Successfully replaced fred with wilma!\n";
}
```

用/g进行全局替换

在前面的例子中你可能注意到了，即使有其他可以替换的部分，s///也只会进行一次替换。当然，这只不过是默认的行为而已。/g修饰符可让s///进行所有可能的、不重复的[注2]替换：

```
$_ = "home, sweet home!";
s/home/cave/g;
print "$_\n";  # 打印"cave, sweet cave!"
```

一个相当常见的全局替换是缩减空白，也就是将任何连续的空白转换成单一空格：

```
$_ = "Input  data\t may have    extra whitespace.";
s/\s+/ /g;  # 现在它变成了"Input data may have extra whitespace."
```

每次只要我们提到如何缩减空白，所有的学生都想知道，如何删除开头和结尾的空白。其实简单到只要两步：

```
s/^\s+//;  # 将开头的空白替换成空字符串
s/\s+$//;  # 将结尾的空白替换成空字符串
```

精简到一步的写法则是使用择一匹配的竖线符号并配合/g修饰符，但这么写其实会运行得稍微慢一点点，至少在编写本书时还是如此。正则表达式的引擎会不断改进，如果要进一步了解如何写出更快（或更慢）的模式，可参阅Jeffery Friedl写的《Mastering Regular Expressions》（O'Reilly）一书：

注2：　不重复是因为每次都会从最近一次替换结束的地方开始新的匹配。

```
s/^\s+|\s+$//g;  # 去除开头和结尾的空白符
```

不同的定界符

就像m//和qw//一样，我们也可以改变s///的定界符。但由于替换运算会用到三个定界符，所以情况又有点不同。

对于一般没有左右之分的（非成对）字符，用法便跟使用斜线一样，只要重复三次即可。下面，我们以井号[注3]作为定界符：

```
s#^https://#http://#;
```

但是，如果使用有左右之分的成对字符，就必须使用两对：一对包住模式，一对包住替换字符串。而且在这种情况下，包住字符串的定界符和包住模式的定界符不必相同。事实上，字符串甚至可以用非成对的定界符。所以下面三行替换操作的写法虽然不同，效果都是一致的：

```
s{fred}{barney};
s[fred](barney);
s<fred>#barney#;
```

可用替换修饰符

除了/g修饰符之外[注4]，我们还可以把用在普通模式匹配中的/i、/x以及/s修饰符用在替换操作中（联合使用的时候无需关心前后顺序）：

```
s#wilma#Wilma#gi;  # 将所有的WiLmA或者WILMA一律替换为Wilma
s{__END__.*}{}s;   # 将__END__标记和它后面的所有内容都删掉
```

绑定操作符

就像在说明m//时提到的，我们可以用绑定操作符为s///选择不同的替换目标：

```
$file_name =~ s#^.*/##s;  # 将$file_name中所有的Unix风格的路径全部去除
```

无损替换

如果需要同时保留原始字符串和替换后的字符串，该怎么办？传统的做法是先复制一份拷贝后再替换：

注3：　对我们的英国朋友致歉，因为井号与英镑都念成“pound”！虽然在Perl里，井号一般用于注释开头，但当解析器知道要等待某个定界符时，井号就不会被视为注释开头。紧跟在代表开始替换的s字符之后的井号，就是其中一个例子。

注4：　即使现在的定界符不是斜线，我们也会将修饰符写成/i。

```
my $original = 'Fred ate 1 rib';
my $copy = $original;
$copy =~ s/\d+ ribs?/10 ribs/;
```

也可以把后面两步并作一步，先做赋值运算，然后针对运算结果进行替换：

```
(my $copy = $original) =~ s/\d+ ribs?/10 ribs/;
```

看起来确实叫人眼花缭乱，因为很多人都会忘记其实左边的赋值运算就好比是普通的字符串，实际做替换的是变量$copy。Perl 5.14增加了一个/r修饰符，专门用于解决这类问题。原先s///操作完成后返回的是成功替换的次数，加上/r之后，就会保留原来字符串变量中的值不变，而把替换结果作为替换操作的返回值返回：

```
use 5.014;

my $copy = $original =~ s/\d+ ribs?/10 ribs/r;
```

形式上看起来和之前的例子差别不大，只不过拿掉了括号罢了。但在这个例子中，运算顺序却是相反的，先做替换，再做赋值。

大小写转换

在替换运算中，常常需要把所替换的单词改写成全部大写（或全部小写）。用Perl很容易就能做到[注5]，只要使用特定的反斜线转义符就行了。\U转义符会将其后的所有字符转换成大写的：

```
$_ = "I saw Barney with Fred.";
s/(fred|barney)/\U$1/gi;  # $_现在成了"I saw BARNEY with FRED."
```

类似地，\L转义符会将它后面的所有字符转换成小写的。沿用前面的例子：

```
s/(fred|barney)/\L$1/gi;  # $_ 现在成了 "I saw barney with fred."
```

默认情况下，它们会影响之后全部的（替换）字符串。你也可以用\E关闭大小写转换的功能：

```
s/(\w+) with (\w+)/\U$2\E with $1/i;  # $_ 替换后为 "I saw FRED with barney."
```

使用小写形式（\l与\u）时，它们只会影响紧跟其后的第一个字符：

```
s/(fred|barney)/\u$1/ig;  # $_ 替换后为 "I saw FRED with Barney."
```

注5： 请重温一下第八章中“选择一种字符解释方式”一节里提到的所有警告。

你甚至可以将它们并用。同时使用\u与\L来表示“后续字符全部转为小写的，但首字母大写”[注6]：

```
s/(fred|barney)/\u\L$1/ig;  # $_ 现在成了 "I saw Fred with Barney."
```

附带一提，虽然这里介绍的是替换时的大小写转换，但它们同样可用在任何双引号内的字符串中：

```
print "Hello, \L\u$name\E, would you like to play a game?\n";
```

split操作符

另一个使用正则表达式的操作符是split，它会根据给定的模式拆分字符串。对于使用制表符、冒号、空白或任意符号分隔不同字段数据的字符串来说，用这个操作符分解提取字段相当方便[注7]。只要你能将分隔符写成模式（通常是很简单的正则表达式），就可以用split分解数据。它的用法如下：

```
my @fields = split /separator/, $string;
```

这里的split操作符[注8]用拆分模式扫描指定的字符串并返回字段（也就是子字符串）列表。期间只要模式在某处匹配成功，该处就是当前字段的结尾、下一个字段的开头。所以，任何匹配模式的内容都不会出现在返回字段中。下面就是典型的以冒号作为分隔符的split模式：

```
my @fields = split /:/, "abc:def:g:h";  # 得到 ("abc", "def", "g", "h")
```

如果两个分隔符连在一起，就会产生空字段：

```
my @fields = split /:/, "abc:def::g:h";  # 得到 ("abc", "def", "", "g", "h")
```

这里有个规则，它乍看之下很古怪，但很少造成问题：split会保留开头处的空字段，却会舍弃结尾处的空字段。例如[注9]：

注6：\L和\u哪个放前面都可以。Larry认为可能有人会把它们颠倒顺序，但一般这么写都是想要首字母大写，其余小写的效果，所以他让Perl能兼容这种情况。说实话，他真是个好心人。

注7 但“以逗号分隔的值”（一般称为CSV文件）除外。用split处理这类文件往往叫人头痛不已。最好还是借助CPAN上专门的模块Text::CSV来做这项工作。

注8：它是一个操作符，尽管它的行为看来像一个函数，而且大家通常都把它叫做函数。不过，技术细节上的差异已超出了本书范围。

注9：这只是默认行为，目的是为了提高效率。若你担心失去结尾处的空字段，只要以-1作为split的第三个参数就能保留它们了。相关细节请参阅*perlfunc*文档。

```
my @fields = split /:/, ":::a:b:c:::"; # 得到 ("", "", "", "a", "b", "c")
```

利用split的/\s+/模式根据空白符分隔字段也是比较常见的做法。该模式把所有连续空白都视作单个空格并以此切分数据：

```
my $some_input = "This  is a \t        test.\n";
my @args = split /\s+/, $some_input; # 得到 ("This", "is", "a", "test.")
```

默认split会以空白符分隔$_中的字符串：

```
my @fields = split; # 等效于 split /\s+/, $_;
```

这几乎就等于以/\s+/为模式，只是它会省略开头的空字段。所以，即使该行以空白开头，你也不会在返回列表的开头处看到空字段。若你想以这种方式来分解用空格分隔的字符串，则可以用一个空格来作为模式：split' ',$other_string。用一个空格来作为模式是split的特殊用法。

一般来说，用在split中的模式就像之前看到的这样简单。但如果你用到更复杂的模式，请避免在模式里使用捕获圆括号，因为这会启动所谓的"分隔符保留模式"（详情请参考*perlfunc*文档）。如果需要在模式中使用分组匹配，请在split里使用非捕获圆括号(?:)的写法，以避免意外。

join函数

join函数不会使用模式，它的功能与split恰好相反：split会将字符串分解为若干片段（子字符串），而join则会把这些片段接合成一个字符串。join函数的用法如下：

```
my $result = join $glue, @pieces;
```

你可以把join的第一个参数理解为胶水（glue），它可以是任意字符串。其余参数则是一串片段。join会把胶水涂进每个片段之间并返回结果字符串：

```
my $x = join ":", 4, 6, 8, 10, 12; # $x 为 "4:6:8:10:12"
```

在这个例子中，我们有5个条目，所以最后接合而成的字符串只有4个冒号。也就是说，有4层胶水。胶水只在两个片段中间出现，不在之前也不在其后。所以胶水的层数会比列表中的条目数少一个。

换句话说，列表至少要有两个元素，否则胶水无法涂进去：

```
my $y = join "foo", "bar";        # 只有一个"bar"，这里不会起作用
my @empty;                        # 空数组
my $empty = join "baz", @empty;   # 没有元素，所以得到一个空字符串
```

使用上面的$x，我们可先分解字符串，再用不同的定界符将它接起来：

```
my @values = split /:/, $x;  # @values为 (4, 6, 8, 10, 12)
my $z = join "-", @values;   # $z为"4-6-8-10-12"
```

虽然split和join合作无间，但请别忘了join的第一个参数是字符串，而不是模式。

列表上下文中的m//

在使用split时，模式指定的只是分隔符：分解得到的字段未必就是我们需要的数据。有时候，指定想要留下的部分反而比较简单。

在列表上下文中使用模式匹配操作符（m//）时，如果模式匹配成功，那么返回的是所有捕获变量的列表；如果匹配失败，则会返回空列表：

```
$_ = "Hello there, neighbor!";
my($first, $second, $third) = /(\S+) (\S+), (\S+)/;
print "$second is my $third\n";
```

如此就能给那些匹配变量起好记的名字，并且下一次模式匹配时仍能使用这些变量。（注意，由于代码中并未用到=~绑定操作符，所以该模式匹配默认是针对$_进行的。）

之前在s///的例子中看到的/g修饰符同样也可以用在m//操作符上，其效果就是让模式能够匹配到字符串中的多个地方。下面的例子中有一对圆括号的模式会在每次匹配成功时返回一个捕获字符串：

```
my $text = "Fred dropped a 5 ton granite block on Mr. Slate";
my @words = ($text =~ /([a-z]+)/ig);
print "Result: @words\n";
# 打印：Fred dropped a ton granite block on Mr Slate
```

这就好比是反过来用split：正则模式指定的并非想要去除的部分，反而是要留下的部分。

事实上，如果模式中有多组圆括号，那么每次匹配就能捕获多个字符串。假设我们想把一个字符串变成哈希，就可以这样做：

```
my $data = "Barney Rubble Fred Flintstone Wilma Flintstone";
my %last_name = ($data =~ /(\w+)\s+(\w+)/g);
```

每次模式匹配成功，就会返回一对被捕获的值，而这一对值正好成为新哈希的键-值对。

更强大的正则表达式

在几乎读了三章有关正则表达式的内容之后，你现在应该明白，这已经成为一项深入Perl核心的强大功能。但不止如此，Perl开发人员还在加入更多功能，你会在接下来的一节看到其中最重要的那些。同时你也会看到正则表达式引擎更有趣的内部运作机制。

非贪婪量词

目前为止，我们（在第七章）看到的4个量词全部都是贪婪（*greedy*）量词。也就是说，在保证整体匹配的前提下，它们会尽量匹配长字符串，实在不行才会吐出一点。比如以`/fred.+barney/`匹配`fred and barney went bowling last night`这个字符串。我们可以一眼就看出来会匹配成功，但现在我们要深入了解一下匹配的过程中到底发生了什么事情[注10]。首先，模式中的`fred`部分将会逐字匹配与其相同的字符串。模式的下个部分是`.+`，它会匹配换行符之外的所有字符（至少一次）。但加号是个贪婪量词，它会尽量匹配最多的字符串。所以，进行到此，它会一口气吞掉字符串剩下的所有内容，一路到最后的`night`。（看到这里你大概觉得要出什么意外了，不过故事还没讲完呢。）

现在轮到模式中的`barney`部分，但它已经没办法进行匹配，因为刚才已经进行到字符串的末尾。由于`.+`就算少匹配一个字符也算匹配成功，所以它打算往后退一步看看，于是吐出最后匹配到的字符`t`。（虽然它很贪婪，不过为了顾全大局，并让整体模式尽可能匹配成功，所以就算自已没有匹配到全部字符串也可以忍受。）

现在又轮到`barney`部分进行匹配，但还是无法成功。因此，`.+`再次吐出一个字符`h`试试看。就这样一个字符一个字符地，`.+`匹配的部分一路减少到了`barney`之前。此刻，模式中的`barney`部分终于能够匹配成功，于是整个模式也就匹配成功了。

正则表达式引擎会一直进行上述的回溯（backtracking）动作，不断地以不同的方式调整模式匹配的内容来适应字符串，直到最终找到一个整体匹配成功的为止，要是直到最后都找不到就宣告失败[注11]。从这个例子我们可以知道回溯的动作可能非常繁琐，因为量词囫囵吞下的字符串太长，于是正则表达式引擎不得不使它逐个吐出来。

注10：正则表达式引擎会进行一定程度的优化，使得事实与这里描述的细节并不完全一样，而优化的方式也随着Perl的改版不断改进。但就功能而言，你不会感觉到事实与描述的有差异。如果你想知道事情的真相，那么你得读最新版的源代码。如果找到缺陷，请提交补丁。

注11：事实上，有些正则引擎会尝试每种可能，就算已经确定匹配成功，也会继续找下去。不过Perl的正则表达式引擎只专注于模式是否能够成功匹配，所以只要找到一个成功匹配的就算完成任务，不再细究下去。具体细节可以参考Jeffrey Friedl的《Mastering Regular Expressions》（O'Reilly）一书。

不过，每个贪婪量词都有一个非贪婪版本。以加号（+）为例，我们可以改用非贪婪的量词+?，这除了表示一次或多次（也就是加号本身的意义）之外，同时要求能匹配的字符串越短越好，而不再是越长越好。现在我们把刚才的模式改成/fred.+?barney/，看看这个新量词是如何运作的。

模式中的fred还是老样子，匹配字符串开头部分。但这次模式中的下一个部分换成了.+?，它会匹配一个以上的字符，但越短越好，也就是最好只匹配一个字符，所以它匹配的部分是fred后面的空白符。接下来出现的模式是barney，但就目前的位置而言，匹配会失败（因为当前位置上开始的字符串是and barney……）。于是.+?模式又很不情愿地多匹配了一个字符a，然后把控制权交给之后的模式重试。但barney还是匹配失败，所以.+?只好再吞下一个字符n，就这样一直进行下去。直到.+?匹配了5个字符之后，barney模式终于能够匹配成功，于是整个模式也就匹配成功了。

其实还是有一些回溯动作的，但这次正则表达式引擎的回溯次数比较少，在速度上应该是大有改善才对。不过，这是在barney跟fred都离得很近的情况下才会成立。如果要处理的数据都是fred在字符串开始处，barney在字符串末尾的话，那么选用贪婪量词反而会比较快。所以最终的速度其实取决于正则表达式要处理的数据。

非贪婪量词并不只跟效率有关。尽管贪婪版本可以成功匹配的字符串，非贪婪版本也同样可以匹配成功（匹配失败的情况亦然），但它们匹配到的字符串长度是不同的。举例来说，假设你手边有一些类似HTML [注12] 的文本，而你需要去除<BOLD>跟</BOLD>这样的标记并保留剩余的内容。如果要处理的字符串是这样：

```
I'm talking about the cartoon with Fred and <BOLD>Wilma</BOLD>!
```

那么可以用下面这个替换表达式把标记去掉。但这会有什么问题呢？

```
s#<BOLD>(.*)</BOLD>#$1#g;
```

问题就出在星号量词太贪心了 [注13] 。如果是下面这个字符串该怎么办？

```
I thought you said Fred and <BOLD>Velma</BOLD>, not <BOLD>Wilma</BOLD>
```

注12：　再次说明，我们所用的并非严格意义上的HTML代码，实际上，我们无法仅仅使用简单的正则表达式就来完成解析HTML的复杂工作。如果你真的需要处理HTML或类似的标记语言，请使用CPAN上的相关模块，比如HTML::Parser，来完成复杂任务。

注13：　还有一个可能会发生的问题：此处应该加上/s修饰符，因为结束标记可能直到后面好几行才会出现。好在此处只是个例子。如果你正在设计产品，那就应该遵照我们的建议，找个质量较高的模块来用。

在此情况下，该模式就会从第一个<BOLD>匹配直到最后一个</BOLD>，把这中间的部分全部取出来。这就错了，我们这里要用的其实是非贪婪量词。非贪婪版本的星号是*?，所以新的替换表达式应该改写成这样：

```
s#<BOLD>(.*?)</BOLD>#$1#g;
```

这么一来就对了。

既然加号的非贪婪版本是+?，星号的非贪婪版本是*?，那么你大概也猜出来了另外两个量词的非贪婪形式也应该相类似。花括号量词的非贪婪版本看起来差不多，只不过问号是加在右括号后面，写成{5,10}?或者{8,}? [注14] 。就连问号量词也有非贪婪的版本：??。虽然这还是一样会匹配到一次或零次，但会优先考虑零次的情况。

跨行的模式匹配

传统的正则表达式都是用来匹配单行文本。由于Perl可以处理任意长度的字符串，其模式匹配自然也可以处理多行文本。这其实和处理单行文本并无本质上的差别。当然了，先得有表达式可以表示多行文本才行。下面的写法可以表示4行的文本：

```
$_ = "I'm much better\nthan Barney is\nat bowling,\nWilma.\n";
```

我们知道，^和$都是代表整个字符串的开头和结尾的锚位（见第八章）。但当模式加上/m修饰符之后，就可以用它们匹配字符串内的每一行（把m看作多行（multiple lines）匹配会比较容易记住），这样一来，它们代表的位置就不再是整个字符串的头尾，而是每行的开头跟结尾了 [注15] 。因此，下面的模式就成立了：

```
print "Found 'wilma' at start of line\n" if /^wilma\b/im;
```

同样地，你也可以对多行文本逐个进行替换。接下来的程序会先把整个文件读进一个变量 [注16] ，然后把文件名作为每一行的前缀进行替换：

```
open FILE, $filename
    or die "Can't open '$filename': $!";
my $lines = join '', <FILE>;
$lines =~ s/^/$filename: /gm;
```

注14：　理论上，即使次数是个明确的数字，也会存在非贪婪量词，例如{3}?。但由于已经直接讲明了次数，所以其实也就失去了贪婪与非贪婪之间的灵活性了。

注15：　记住，这就是为什么我们之前建议你改用更为严谨的\A和\z锚位的原因。如果你确实要匹配字符串的开头和结尾的话，就应该用这两个锚位。

注16：　希望它很小。我们指的是文件，而不是变量。

一次更新多个文件

通过程序自动更新文件内容时，最常见的做法就是先打开一个和原来内容一致的新文件，然后在需要的位置进行改写，最后把修改后的内容写进去。后面会看到，这样做和直接更新文件的做法的效果大致相同，只是有些附带的好处。

比如下面这个例子，假设现在有几百个格式类似的文件。其中一个叫做*fred03.dat*，里面都是类似下面这几行的内容：

```
Program name: granite
Author: Gilbert Bates
Company: RockSoft
Department: R&D
Phone: +1 503 555-0095
Date: Tues March 9, 2004
Version: 2.1
Size: 21k
Status: Final beta
```

我们想要修改这个文件，让它有一些不同的信息。下面大致就是最终改好后的样子：

```
Program name: granite
Author: Randal L. Schwartz
Company: RockSoft
Department: R&D
Program name: granite
Author: Randal L. Schwartz
Company: RockSoft
Department: R&D
Date: June 12, 2008 6:38 pm
Version: 2.1
Size: 21k
Status: Final betaDate: June 12, 2008 6:38 pm
Version: 2.1
Size: 21k
Status: Final beta
```

简单地说，我们要做三项改动：`Author`字段的姓名要改，`Date`要改成今天的日期，而`Phone`则要删除。另外还有几百个文件也都要进行类似的修改。

要在Perl中直接修改文件内容可以使用钻石操作符（`<>`）。虽然一眼可能看不出端倪，但下面的程序确实可以完成刚才的要求。此程序的新意在于特殊变量`$^I`。现在我们先跳过它，等一下再说明：

```
#!/usr/bin/perl -w

use strict;

chomp(my $date = `date`);
```

```
$^I = ".bak";

while (<>) {
    s/^Author:.*/Author: Randal L. Schwartz/;
    s/^Phone:.*\n//;
    s/^Date:.*/Date: $date/;
    print;
}
```

因为我们需要今天的日期，所以这个程序一开始就使用了系统的*date*命令。此外也可以在标量上下文中使用Perl自己的localtime函数（但两者的格式稍有不同），性能更好些：

```
my $date = localtime;
```

下一行则是对$^I变量赋值，不过我们现在先跳过不看。

钻石操作符会读取命令行参数指定的那些文件。程序的主循环一次会读取、更新及输出一行（以之前学到的知识来推断，你没准觉得经过修改的内容会飞快地在终端上滚过，而本来的文件却没修改。不过请耐心地看下去）。注意第二个替换运算是把电话号码那一行换成空字符串，连换行符也一起去掉。所以到了要输出的时候，其实什么都不会输出，就好像从来就没有出现过Phone这个字段一样。由于大部分的输入行都不会匹配这三个模式，所以它们在输出的时候都不会有任何变动。

这样的结果跟我们想要的已经相差不远了，但我们还没有告诉你更新过的内容是如何写回文件的。这个问题的答案就在$^I这个变量中。这个变量的默认值是undef，也不会对程序造成任何影响。但如果将其赋值成某个字符串，钻石操作符（<>）就会具有比平常更多的魔力。

对于钻石操作符的魔力我们已经知道不少：它会自动帮你打开许多文件，而且如果没有指定文件，它就会从标准输入读进数据。但如果$^I中是个字符串，该字符串就会变成备份文件的扩展名。现在我们来看看这是如何运作的。

先假设钻石操作符正好打开了文件*fred03.dat*。除了像以前一样打开文件之外，它还会把文件名改成*fred03.dat.bak* [注17] 。虽然打开的是同一个文件，但是它在磁盘上的文件名已经不同了。接着，钻石操作符会打开一个新文件并将它取名为*fred03.dat*。这么做并不会有任何问题，因为我们已经没有同名文件了。现在钻石操作符会把默认的输出设定为这个新打开的文件，所以输出来的所有内容都会被写进这个文件 [注18] 。这样while循

注17：　在非Unix系统上实现细节可能有些不同，但结果应该都是一样的。详见你的Perl版本的注意事项。

注18：　钻石操作符也会尽可能复制原文件的使用者权限以及所有者设定。例如，如果本来的文件是所有用户皆可读取，那么新的文件也应该如此。

环会从旧文件读进一行输入，做了一些改动之后把新的内容写进新文件。在普通的机器上，这样的程序可以在几秒内更新好上百个文件。够厉害吧？

所以程序结束后，用户会发现什么呢？他们会说“喔，我懂了。Perl根据我的需要编辑了*fred03.dat*文件的内容，而且好心地把原始拷贝备份到*fred03.dat.bak*文件”。不过我们知道真相，Perl并没有编辑任何文件，它只是创建了一个修改过的拷贝。趁我们还在盯着他那冒烟的魔术师手杖看的时候，把文件偷偷调了包。真是高明呀！

有些人会把`$^I`的值设为~这个字符，因为*emacs*在处理备份文件的文件名时也是这么做。而如果把`$^I`设为空字符串，就会直接修改文件的内容，但不会留下任何备份。只要模式中不小心打错一个字，就可能会把整份数据全都清空，所以如果你真的想看看备份磁带质量如何，就尽情地用空字符串吧！全部确认无误之后再把备份文件删除是轻而易举的事。如果做错了，则需要把已备份的文件还原回来，大概你已经想到可以用Perl来做这件事了（参见第十三章的“重命名文件”一节中的例子）。

从命令行直接编辑

之前一节中通过编程修改文件内容的方式已经非常简单了，不过，Larry认为那还不够。

假设你需要更新上百个文件，把里面拼错成`Randall`的名字改成只有一个`l`的`Randal`。你可以写个和之前类似的程序完成此事。或者，你也可以在命令行上使用如下单行程序一步完成：

```
$ perl -p -i.bak -w -e 's/Randall/Randal/g' fred*.dat
```

Perl的命令行选项设计非常巧妙，让你只用极少的按键就能建立一个完整的程序[注19]。我们先来看看这个例子中的选项的用处。

以*perl*开头的命令的作用如同在文件的开头写上`#!/usr/bin/perl`：表示使用*perl*程序来处理随后的脚本。

`-p`选项则可以让Perl自动生成一小段程序，看起来类似如下片段[注20]：

```
while (<>) {
  print;
}
```

如果不需要这么多功能，还可以改用`-n`选项，这样可以把自动执行的`print`去掉，所以你可以自行决定什么内容需要（*awk*的粉丝们会比较熟悉`-p`和`-n`选项）打印。这点细微差

注19：　参考*perlrun*文档，其中列有全部可用选项。

注20：　其实`print`出现在`continue`块中。进一步的信息请参阅*perlsyn*和*perlrun*文档。

别对大程序来说无关轻重，但对于节约按键时间来说还是很有好处的。

下一个出现的选项是-i.bak，其作用就是在程序开始运行之前把$^I设为".bak"。如果你不想做备份，请直接写出-i，不要加扩展名。如果你不要备用的降落伞，那就只带一副上飞机吧。

之前已经介绍过-w选项了，它能开启警告功能。

选项-e用来告诉Perl后面跟着的是可供执行的程序代码。也就是说，s/Randall/Randal/g这个字符串会被直接当成Perl程序代码。因为目前我们已经有个while循环（来自-p选项）了，所以这段程序代码会被放到循环中print前面的位置。基于一些技术上的原因，用-e选项指定的程序中可以省略末尾的分号。如果你指定了多个-e选项，就会有多段程序代码，此时只有最后一段程序末尾的分号可以省略。

最后一个命令行参数是fred*.dat，表示@ARGV的值应该是匹配此文件名模式的所有文件名。把以上所有片段全都组合在一起，就好像写了下面这个程序并且用fred*.dat这个参数调用它一样：

```
#!/usr/bin/perl -w

$^I = ".bak";

while (<>) {
    s/Randall/Randal/g;
    print;
}
```

把这个程序与我们上一节使用的程序相比，会发现这两者十分相似。通过命令行选项就能完成这么一大堆事情，写起来又轻便，是不是很赞？

习题

以下习题答案参第321页上的“第九章习题解答”一节：

1. [7]建立一个模式，无论$what的值是什么，它都可以匹配三个$what的内容连在一起的字符串。也就是说，如果$what的值是fred，那么你的模式应该匹配fredfredfred；若$what的值为fred|barney，那么你的模式应该匹配fredfredbarney、barneyfredfred、barneybarneybarney或许多其他组合。（提示：你应该在模式测试程序的开头放上类似my$what='fred|barney';这样的语句。）

2. [12]写个程序来复制并修改指定的文本文件。在副本里，此程序会把出现字符串Fred（大小写不计）的每一处都换成Larry（也就是Manfred Mann换成ManLarry Mann）。输入文件名应该在命令行上指定（不询问用户），输出文件名则是本来的文件名加上.out。

3. [8]修改前一题程序，把所有的Fred换成Wilma并把所有的Wilma换成Fred。如果输入的是fred&wilma，那么正确的输出应是Wilma&Fred。

4. [10]附加题：写个程序，把你目前写过的所有程序都加上版权声明，也就是加上一行这样的文字：

   ```
   ## Copyright (C) 20XX by Yours Truly
   ```

   把它放在“shebang”行之后。你应该直接修改文件内容并且做备份。假设你将在命令行指定待修改文件的名称。

5. [15]额外附加题：修改前一题程序里的模式，如果文件里已经有版权声明，就不再进行修改。提示：你可能需要知道钻石操作符当前正在读取的文件的名称，可以在$ARGV里找到。

第十章

其他控制结构

本章你会看到其他编写Perl程序的方式。总的来说，这些技术并不会提升Perl语言本身的威力，但用来完成任务、解决问题还是非常轻松容易的。你不一定要在自己的程序中使用这些技术，不过可别因此小看这些内容，迟早你都会在别人的程序代码中看到这些控制结构（事实上，读完本书前，你肯定会看到这些技术的实际用例）。

unless控制结构

在if控制结构中，只有当条件表达式（conditional expression）为真时，才执行某块代码。如果你想让代码块在条件为假时才执行，请把if改成unless：

```
unless ($fred =~ /\A[A-Z_]\w*\z/i) {
    print "The value of \$fred doesn't look like a Perl identifier name.\n";
}
```

使用unless意味着，除非（unless）执行条件为真，否则就执行里面的代码。这就好像使用if测试来判断相反的条件。另一种说法是它类似于独立的else子句。也就是说，当看不懂某个unless语句时，你总可以用下面这样的if测试等价表示（心里默默转换也好，实际改写也罢）：

```
if ($fred =~ /\A[A-Z_]\w*\z/i) {
    # 什么都不做
} else {
    print "The value of \$fred doesn't look like a Perl identifier name.\n";
}
```

这么做与运行效率高低无关，两种写法应该会被编译成相同的内部字节码。另外一个改写的方法，就是以否定操作符（!）来否定条件表达式：

```
if ( ! ($fred =~ /\A[A-Z_]\w*\z/i) ) {
    print "The value of \$fred doesn't look like a Perl identifier name.\n";
}
```

一般来说，我们应该选择最容易理解的方法写代码。对程序维护员来说，读得越顺，理解起来也就越容易。如果用if表达比较拗口，要加上否定才通顺的话，那就应该改用unless。以后你就会发现，有时候只有使用unless才更为自然。

伴随unless的else子句

其实unless之后也可以加上一个else子句。虽然这在语法上是允许的，但却有潜在的语义混淆：

```
unless ($mon =~ /\AFeb/) {
    print "This month has at least thirty days.\n";
} else {
    print "Do you see what's going on here?\n";
}
```

确实有人希望能这么写，特别是当第一个子句相当短（也许只有一行）而第二个子句又有很多行的时候。不过我们可以把它改写成否定的if语句，或者干脆对调两个子句成为一个普通if控制结构：

```
if ($mon =~ /\AFeb/) {
    print "Do you see what's going on here?\n";
} else {
    print "This month has at least thirty days.\n";
}
```

有一点很重要，请记住，代码的读者永远可以分为两类：执行代码的机器以及维护代码的人类。如果人类都无法理解你写的程序，那迟早机器也会做错事情。

until控制结构

有时也许你会想要颠倒while循环的条件表达式。那么，请使用until语句：

```
until ($j > $i) {
    $j *= 2;
}
```

这个循环会一直执行，直到条件为真。它只不过是个改装过的while循环罢了，两者之间的唯一差别在于，until会在条件为假时重复执行，而不是为真时执行。因为条件判断发生在循环第一次迭代之前，所以它仍旧是一个执行零次以上的循环，和while循环一样[注1]。

注1：Pascal程序员请注意：你们的repeat-until循环至少会执行一次，但Perl的until循环可能一次也不执行，因为条件判断是在每次循环执行之前进行的。

类似if和unless转化的例子，你可以用否定条件表达式的方法，把任意一个until循环改写成while循环。不过随着时间推移，你会逐渐习惯采用简单而自然的写法来使用until控制结构。

表达式修饰符

为了进一步简化代码书写，表达式后面可以接一个用于控制它行为的修饰符。比如下面这个if修饰符，它实际的作用相当于一个if语句块：

```
print "$n is a negative number.\n" if $n < 0;
```

这其实能达到和下面代码完全相同的效果，但我们省去了键入圆括号和花括号的工作[注2]：

```
if ($n < 0) {
    print "$n is a negative number.\n";
}
```

之前曾经提到，那些使用Perl的家伙都懒于打字。不过这个更短的版本其实也更容易用英语读出来：print this message if $n is less than zero。

注意，即使条件表达式写在后面，它仍然会先执行。这与通常由左至右的顺序相反。解读Perl代码的方法就是像Perl的内部编译器一样，先把语句全部读完再判断其含义。

还有其他几个修饰符：

```
&error("Invalid input") unless &valid($input);
$i *= 2 until $i > $j;
print " ", ($n += 2) while $n < 10;
&greet($_) foreach @person;
```

以上写法都能按照我们原本的意图正常运作。换句话说，上面每一行都可以效仿if修饰符范例进行改写。比如第三条可以改写成：

```
while ($n < 10) {
    print " ", ($n += 2);
}
```

值得注意的是，在print参数列表中，圆括号里的表达式会将$n加2，并将结果存回$n，然后返回最新的值并打印出来。

注2： 当然还省去了换行符。但是要注意花括号其实还有创建新的变量作用域的功能。在某些时候这很有用，要进一步了解细节，还请参阅文档。

这些简写形式读起来像自然语言：调用call the &greet subroutine for each @person in the list(调用&greet子程序问候@person中的每个成员);double $1 until it's large than $j(倍增$i直到它大于$j) [注3] 。这些修饰符的常见用法之一，就是写成下面这样的语句：

```
print "fred is '$fred', barney is '$barney'\n" if $I_am_curious;
```

用这种“倒装句”编写程序可以把语句中重要的部分放在前面。上面那个语句的重点是查看一些变量的值，而不是检查你是否好奇 [注4] 。有些人喜欢将整个语句写成一行，也可能在if之前加上些制表符使它向右边缩进一些，上面那个例子就是如此。也有人喜欢将if修饰符放在下一行并缩进一段距离：

```
print "fred is '$fred', barney is '$barney'\n"
    if $I_am_curious;
```

虽然这些带有修饰符的表达式都可以用块的形式重写，也就是用最传统的方法写，但反方向的重写却不一定成功。修饰符的两边都只能写单个表达式，因此不能写某事if某事while某事until某事unless某事，因为那样太让人困惑了。另外修饰符的左边也不能放多条语句。如果确实需要，还是建议你回到传统做法，仍然写那些圆括号和花括号。

如同我们之前提到if修饰时所说的，右边的控制表达式总是先求值，和老式写法的执行顺序是一样的。

在使用foreach修饰符的时候无法自选控制变量，必须得使用$_。这通常不是问题，不过若真需要自选控制变量，可以用老式的foreach循环改写。

裸块控制结构

所谓的“裸块（naked block）”就是没有关键字或条件表达式的代码块。好比现在有一个while循环，如下所示：

```
while (condition) {
    body;
    body;
    body;
}
```

然后拿走关键字while和条件表达式，就会得到一个裸块：

注3：　好吧，至少我们是这么理解阅读的。

注4：　当然，$I_am_curious这个变量是我们杜撰出来的，并非内置的Perl变量。通常使用这个技巧的人会将变量命名为$TRACING，或者用constant编译指令来声明一个全局常量。

```
{
    body;
    body;
    body;
}
```

裸块就像一个while或foreach循环，只是它从不循环，它仅仅执行循环体一次，然后结束。所以，裸块其实并非循环！

稍后就能看到裸块的一些其他应用，这里先看它是如何为临时词法变量限定作用域的：

```
{
    print "Please enter a number: ";
    chomp(my $n = <STDIN>);
    my $root = sqrt $n;  # 计算平方根
    print "The square root of $n is $root.\n";
}
```

这个块中的$n和$root都是限于此块的临时变量。一个关于局部变量的准则是：最好把变量声明在最小使用范围之内。如果某个变量只会在几行代码里使用，你可以把这几行放到一个裸块里并就近声明变量。当然，如果我们稍后还需要用到$n或者$root之类的变量，便需要在更大范围中声明这些变量。

你可能已经注意到这里的sqrt函数很陌生。没错，那是一个我们不曾见过的函数。Perl有许多内置函数无法在本书一一介绍，请查阅*perlfunc*文档自行学习。

elsif子句

许多情况下，你需要逐项检查一系列的条件表达式，看看其中哪个为真。这可以通过if控制结构的elsif子句完成此事，比如下面的代码：

```
if ( ! defined $dino) {
    print "The value is undef.\n";
} elsif ($dino =~ /^-?\d+\.?$/) {
    print "The value is an integer.\n";
} elsif ($dino =~ /^-?\d*\.\d+$/) {
    print "The value is a _simple_ floating-point number.\n";
} elsif ($dino eq '') {
    print "The value is the empty string.\n";
} else {
    print "The value is the string '$dino'.\n";
}
```

Perl会一个接一个地测试这些条件表达式。当其中某项符合时，相应的程序代码块就会

被执行，然后整个控制结构结束[注5]，并执行接下来的程序代码。如果没有任何一项符合，则执行最末端的else块（当然，else子句无疑是可以省略的，但这里最好保留，方便说明）。

elsif子句的数量并没有限制，但别忘了Perl必须执行前面的99个失败的测试，才会到达第100个。如果要写十几个elsif，不妨考虑使用更加有效的方式来编写。Perl常见问题集（参阅*perlfaq*文档）列出了一堆关于如何模拟case或switch的建议，Perl 5.10或者更高版本的用户还可以选择使用第十五章中介绍的given-when结构作为变通方式。

你可能已经注意到了，这个关键字的拼写居然是elsif，好像缺少了一个e。但如果你写成具有两个e的elseif，Perl会告诉你拼写错误。为什么呢？因为Larry说了算[注6]。

自增与自减

编程中常常需要对标量变量的值进行自增或自减。因为这种需求太普遍了，所以像其他常用表达式一样，有相应的简写。

使用自增操作符（autoincrement operator，++）能将标量变量加1，就像C语言及相似程序语言中的相同操作符那样：

```
my $bedrock = 42;
$bedrock++;  # $bedrock加1，变成43
```

和其他将变量加1的方法一样，若有需要标量将会被自动创建：

```
my @people = qw{ fred barney fred wilma dino barney fred pebbles };
my %count;                      # 新的空哈希
$count{$_}++ foreach @people;   # 按需要创建新的键-值对
```

第一次处理foreach循环时，$count{$_}会自增。先是$count{"fred"}，因此它会从undef成为1，因为之前这个哈希值不存在。下一次执行循环时，$count{"barney"}会变成1；在这之后，$count{"fred"}会变成2。每次处理循环时，%count中的某个元素就会自动递增，当然也有可能被创建。在整个循环完成后，$count{"fred"}的值应该是3。这是个快速而简易的方法，可用来检查列表中有哪些元素并计算每个元素出现的次数。

注5：　不像C语言的“switch”结构那样会接着执行下一个块。

注6：　事实上，他坚持这种拼法，他甚至拒绝融入另一种拼法的建议。如果你坚持要拼写另一个e也很简单：第一步，发明你自己的语言；第二步，让语言变得非常流行。如果程序语言是你自己发明的，关键字要怎么拼是你的自由。我们希望你的语言不会是第一个有“elseunless”关键字的语言。

类似地，自减操作符（autodecrement operator，--）会从标量变量中减去1：

```
$bedrock--;  # $bedrock 减1，又变回42了
```

自增的值

我们可以在取得变量值的同时修改变量值。把++操作符写在变量之前就能先增加变量的值，然后获取新值。我们把这种操作称为前置自增（*preincrement*）：

```
my $m = 5;
my $n = ++$m;  # 先增加$m的值到6，再把该值放入$n
```

或者把--操作符放在变量之前，先自减，再取新值。我们把这种操作称为前置自减（*predecrement*）：

```
my $c = --$m;  # 先减少$m的值到5，再把该值放入$c
```

接下来是比较有特别的部分。将变量名称放在前面就表示先取值，然后再自增或自减。这样的操作我们称为后置自增（postincrement）或后置自减（postdecrement）：

```
my $d = $m++;  # $d得到的是$m之前的值（5），然后$m增加到6
my $e = $m--;  # $e得到的是$m之前的值（6），然后$m减少到5
```

之所以说它特别，是因为这里同时做了两件事。我们在同一个表达式中取值并且修改它的值。如果操作符在前，就会先自增（或是自减），然后使用新值；如果变量在前，就会先返回其原来的值，然后再自增或自减。另外一种说法是，这些操作符会返回某个变量值，但顺便它们还具有修改变量值的副作用。

如果表达式中只由这种操作符组成[注7]，不提取变量值，而只是利用修改值的副作用的话，那么操作符前置或后置都一样，没有任何区别[注8]：

```
$bedrock++;  # $bedrock加1
++$bedrock;  # 同样，$bedrock加1
```

这类操作符的一个常见用法就是配合哈希计数判断之前已见过的条目：

```
my @people = qw{ fred barney bamm-bamm wilma dino barney betty pebbles };
my %seen;

foreach (@people) {
    print "I've seen you somewhere before, $_!\n"
```

注7：　也就是空上下文。

注8：　参与过程序语言内部实现的人可能会猜测，后置自增及后置自减是否会比它们的前置版本要慢，但 Perl 不会死脑筋，在空上下文中，Perl会自动对后置形式进行优化。

```
        if $seen{$_}++;
}
```

当barney第一次出现的时候，$seen{$_}++的值为假，因为$seen{$_}的值也就是$seen{"barney"}的值，为undef。不过由于这个表达式具有将$seen{"barney"}递增的副作用，所以当再次遇到barney的时候，$seen{"barney"}的值就是真，使得程序输出信息。

for控制结构

Perl的for控制结构类似其他语言（如C语言）当中的常见for循环。大体结构看起来就像这样：

```
for (初始化; 测试; 递增) {
    程序主体;
    程序主体;
}
```

虽然对Perl而言，这种类型的循环事实上只是一种变相的while循环，如下所示 [注9]：

```
初始化;
while (测试) {
    程序主体;
    程序主体;
    递增;
}
```

for循环目前最常见的用途，就是控制重复的运算过程：

```
for ($i = 1; $i <= 10; $i++) {  # 从1数到10
    print "I can count to $i!\n";
}
```

如果之前见过这种用法，那么你不看注释也会知道第一行在说什么。在循环开始前，控制变量$i被设置为1。然后，它就像while循环一样，当$i的值小于或等于10时，循环会不断迭代执行。每次迭代之后与下一次迭代之前会进行递增运算，也就是将控制变量$i加1。

因此，在循环第一次执行时，$i是1。因为它小于或等于10，所以程序会输出信息。虽然递增操作符被写在循环顶端，但在逻辑上它却是位于循环底部，等输出信息之后才会执行。于是，$i递增到2，依然小于或等于10，因此程序会再次输出信息。接着$i递增到3，还是小于或等于10，依此类推。

最后，程序会输出数到9的信息。然后$i递增到10，依然小于或等于10，所以程序会执行最后一次循环，并且输出信息表明数到了10。$i在最后一次递增时变成了11，这次不再

注9：　其实递增是在continue块里发生的，本书不会涉及。请查看*perlsyn*文档了解真相。

小于或等于10。所以控制权退出循环之外，继续执行接下来的代码。

因为这三个部分被一起放在循环的开头，所以老练的程序员看到第一行就明白："这是一个将$i从1数到10的循环。"

注意，当循环结束之后，它的控制变量会在范围之外。在这个例子里面，控制变量的值已经涨到了11 [注10] 。这种循环非常灵活，可以用来进行各式各样的计数。比如从10倒数到1：

```
for ($i = 10; $i >= 1; $i--) {
    print "I can count down to $i\n";
}
```

这个循环从-150开始累加3，一直加到1000 [注11] ：

```
for ($i = -150; $i <= 1000; $i += 3) {
    print "$i\n";
}
```

事实上这三个循环控制部分（初始化、测试和递增）都可以为空，但即使不需要它们也还得保留分号。在下面这个不太常见的例子里，测试部分是一个替换运算，而递增部分则是空：

```
for ($_ = "bedrock"; s/(.)//; ) {  # 当s///这个替换成功时，循环继续
    print "One character is: $1\n";
}
```

在隐式while循环中的测试表达式是一个替换运算，成功替换时会返回真。在这个例子里，当循环第一次执行时，替换运算会拿走bedrock中的b字母。每次执行循环会拿走一个字母，直到字符串为空。这时替换运算会失败，导致循环结束。

若（在两个分号之间的）测试表达式为空，它会被自动当成真值，从而导致无限循环。不过在你学到如何中断这种循环之前请先不要造出无限循环，稍后我们就会告诉你如何中断无限循环：

```
for (;;) {
    print "It's an infinite loop!\n";
}
```

注10： 请务必看看《This is Spinal Tap》这部风靡一时的电影，了解什么是"up to eleven"。

注11： 注意，其实是不可能真的数到1000的，最后一个迭代时小于1 000的数字是999，因为$i所有的值都应该是3的整数倍。

如果真的需要，更具Perl风格的无限循环[注12]是while版本的：

```
while (1) {
    print "It's another infinite loop!\n";
}
```

虽然C程序员比较熟悉第一种方式，但即使是初学Perl的人也知道1总是真，从而导致无限循环，因此第二种写法更好一些。Perl很聪明地意识到这种常量表达式是可以优化的，因此不会导致性能问题。

foreach和for间的秘密关系

也许你不知道，在Perl解析器里，foreach和for这两个关键字实际上是等价的。也就是说，当Perl看到其中之一时，就好像看见了另一个。Perl可以从圆括号里的内容判断出你的意图。如果里面有两个分号，它就是之前介绍的for循环；若没有分号，就说明它是一个foreach循环：

```
for (1..10) {  # 实际上就是一个从1到10的foreach循环
    print "I can count to $_!\n";
}
```

这实际上就是一个foreach循环，但用的却是for关键字。除此以外，本书其他例子都会写成foreach的形式。不过在真实世界中，你觉得大多数Perl一族会愿意多打那多出来的4个字母么[注13]？除了初学者的程序代码，它通常都会被写成for，所以你必须像Perl一样通过检查分号来判断它是哪一种循环。

在Perl世界里，纯正的foreach循环几乎总是更好的选择。在上面的foreach循环例子（表面上写成了for循环）里面，可以一眼看出它是从1到10的循环。可是，下面这个for循环也在试图做一样的事情，你看得出它有什么问题么？请自己找答案，不要偷看脚注[注14]：

注12：　如果不小心造成了一个无限循环，试试看按Control+C能否终止。很可能在你按下之后，程序还是会在屏幕上滚过一些信息才停止，这是系统I/O和其他因素导致的。嘿，我们已经警告过你了。

注13：　要是你真的这么想，说明你还是初学者，仍然没什么人关注你。在程序员（尤其是Perl程序员）看来，懒惰是传统美德。如果不信，下次参加Perl Monger聚会的时候可以调查一下。

注14：　这里有两个半错误。第一，条件部分使用小于号，因此实际的循环只执行9次而非10次。这种循环很容易陷入所谓的“栅栏柱（fencepost）”问题，比如有位农夫想建一条30米的栅栏，每隔3米立一个柱子，那么农夫需要多根栅栏柱才够？答案可不是10个哦。第二，控制变量是$i，但循环体中使用的却是$_。还有半个错误是，对于这种写法，需要读、写、维护、调试的代码更多，这也就是为什么我们说纯正的foreach在Perl里是一个更好的选择。

```
for ($i = 1; $i < 10; $i++) {  # 糟糕！这里某个地方有错误！
    print "I can count to $_!\n";
}
```

循环控制

目前为止你大概已经感觉到，Perl是一种所谓的“结构化”编程语言。特别是Perl程序的任何块都只有一个入口，也就是块的顶端。不过相比前面介绍过的结构，有时候需要更多样化的控制方式。例如你有时候可能需要一个跟while循环相似的循环，但要求它至少执行一次；或者，你偶尔需要提早退出某个代码块。Perl有三个循环控制操作符，你可以在循环里使用，让循环做到各种招式。

last操作符

last操作符能立即中止循环的执行，就像在C这类语言中的“break”操作符一样。它是循环的“紧急出口”。当你看到last，循环就会结束。例如：

```
# 打印输入中所有提到 fred 的行，直到碰到 __END_ 记号为止
while (<STDIN>) {
    if (/__END__/) {
        # 碰到这个记号说明再也没有其他输入了
        last;
    } elsif (/fred/) {
        print;
    }
}
```

只要输入行中有__END__记号，这个循环就会结束。当然，结尾的那行注释只是提醒而已，完全可以省略。我们只是将它放在那里，好让整个过程更加清晰。

在Perl中有5种循环块，它们是for、foreach、while、until以及裸块[注15]。而if块或子程序带的花括号[注16]不是循环块。如同前面的例子，last操作符对整个循环块起作用。

last操作符只会对当前运行中的最内层的循环块发挥作用。要跳出外层块，请继续看下去，我们很快就会提到。

注15：　没错，你可以用last跳出裸块（jump out of a naked block）。但这并不意味着你在家门口裸舞（jumping naked out into your block）。

注16：　这可能是个坏主意，但确实可以在子程序里用循环控制操作符来控制子程序外面的循环。也就是说，如果在循环块内调用一个子程序，并且子程序执行last操作，同时子程序内并没有运行中的循环块，那么程序的流程会跳到主程序中的循环块的后面。在将来的Perl版本中，这种在子程序内的循环控制能力会被去掉，没有人会怀念它的。

next操作符

有时候你并不需要立刻退出循环，但需要立刻结束当前这次循环迭代。这就是next操作符的用处，它会跳到当前循环块的底端[注17]。在next之后，程序将会继续执行循环的下一次迭代，这和C这类语言中的“continue”操作符的功能相似：

```
# 分析输入文件中的单词
while (<>) {
    foreach (split) {  # 将$_分解成单词，然后每次将一个单词赋值给$_
        $total++;
        next if /\W/;    # 如果碰到不是单词的字符，跳过循环的剩余部分
        $valid++;
        $count{$_}++;    # 分别统计每个单词出现的次数
        ## 上面的 next 语句如果运行，会跳到这里 ##
    }
}

print "total things = $total, valid words = $valid\n";
foreach $word (sort keys %count) {
    print "$word was seen $count{$word} times.\n";
}
```

这个例子比前面的要复杂些，所以我们逐步进行解说。while循环逐行读取来自钻石操作符的输入并放进$_，这我们先前已经看过了。循环每次执行时，$_就得到输入数据的下一行。

在循环中，foreach循环能遍历split返回的列表。你还记得split不带参数的默认行为么[注18]？它会用空白来切分$_，也就是说把$_分解成由单词组成的列表。既然foreach循环没有提到其他控制变量，控制变量就应该是$_。因此我们会在$_中依次看到所有单词。

可是，我们不是才说过$_是用来存储每一行的输入么？在外层循环是这样没错，但在foreach循环里，它却能存储每一个单词。Perl能正确处理为了新用途而重用$_，这种事并不奇怪。

对foreach循环来说，每当我们在$_中看到一个单词时，$total就会递增，所以它会是全部单词的总数。但下一行（是这个例子的关键）会检查单词里是否包含任何非单词字符（字母、数字和下划线以外的任何字符）。因此如果其中出现了像Tom's、full-sized，或者后面紧接着逗号、引号或任何其他奇怪的字符的单词，那它就会匹配这个模式，导致跳过循环的其余部分，继续处理下一个单词。

注17：　这又是另一个我们说谎的地方。事实上next会跳到循环中常常省略的continue块的开头处。参见*perlsyn*文档了解详细的信息。

注18：　如果忘掉了也别太担心。不必在那些在*perldoc*文档中能查到的东西上花费太多精力。

不过如果找到了一个普通的单词，比如fred。在此情况下，我们会将$valid加1，连带$count{$_}也累加以记录每个不同单词出现的次数。所以，在这两个循环执行完毕后，我们就完成了对用户指定的所有文件中的每一行里每个单词的计数。

我们不会再解释最后几行的意思。到了这里，我们希望你已经有能力对付这样的程序代码。

跟last一样，next也可以用在5种循环块中：for、foreach、while、until或是裸块。同样地，如果有多层的嵌套循环块，next只会对最内层起作用。这一节的最后，我们将会看到如何突破这种限制。

redo操作符

循环控制操作符的第三个成员是redo。它能将控制返回到当前循环块的顶端，而不经过任何条件测试，也不会进入下一次循环迭代。而那些用过C这类语言的人却会对这个操作符感觉陌生。因为那些语言里没有这个概念。来看具体的例子：

```
# 打字测验
my @words = qw{ fred barney pebbles dino wilma betty };
my $errors = 0;

foreach (@words) {
    ## redo 指令会跳到这里 ##
    print "Type the word '$_': ";
    chomp(my $try = <STDIN>);
    if ($try ne $_) {
        print "Sorry - That's not right.\n\n";
        $errors++;
        redo;  # 跳回循环的顶端
    }
}
print "You've completed the test, with $errors errors.\n";
```

和另外两个操作符一样，redo在5种循环块里都可以使用，并且在循环块嵌套的情况下只对最内层的循环起作用。

next和redo两者之间最大的区别在于，next会正常继续下一次迭代，而redo则会重新执行这次的迭代。下面的程序可以让你体验这三种操作符在工作方式上的区别：

```
foreach (1..10) {
    print "Iteration number $_.\n\n";
    print "Please choose: last, next, redo, or none of the above? ";
    chomp(my $choice = <STDIN>);
    print "\n";
    last if $choice =~ /last/i;
    next if $choice =~ /next/i;
    redo if $choice =~ /redo/i;
    print "That wasn't any of the choices... onward!\n\n";
```

```
}

print "That's all, folks!\n";
```

如果不键入任何字，只是按下回车键，则循环会逐次增加计数。如果你在数字4显示的时候选择last，那么循环就会因此而结束，你将看不到数字5；如果你在数字4显示的时候选择next，就会直接跳到数字5而不打印“onward”信息；如果你在数字4显示的时候选择redo，那么会回到4这个数字重来。

带标签的块

当你需要从内层对外层的循环块进行控制时，请使用标签（label）。在Perl里，标签和其他标识符一样，是由字母、数字和下划线组成的，但不能以数字开头。然而由于标签没有前置符号，就可能和内置函数或自定义子程序名混淆。所以将标签命名为print或if是很糟糕的选择。因此，Larry建议用全大写字母命名标签，这样不仅能防止它跟其他标识符相互冲突，也使得它在程序中能突显出来。无论大写还是小写，标签总是罕见的，只会在很少的Perl程序中出现。

要对某个循环块加上标签，通常只要将标签及一个冒号放在循环前面就行了。之后在循环里，若有需要就可以在last、next或redo的后面加上这个标签：

```
LINE: while (<>) {
  foreach (split) {
    last LINE if /__END__/;  # 跳出标签为LINE的循环
    ...
  }
}
```

为了增进可读性，通常的建议是把标签靠左写，哪怕当前的代码的层次缩进得很深。注意，标签应该用来命名整块代码，而不是用来标明程序中的某个具体位置[注19]。在上面的例子里，特殊的__END__记号代表了输入的结束。只要看到这个记号，程序就会忽略所有接下来的输入行，即使还有其他未读的文件。

通常应该以名词来为循环命名[注20]。也就是说，因为外层循环是每次处理一行，所以可称之为LINE。如果也要为内层循环取个名字，我们可能会叫它WORD，因为它每次处理一个单词。如此一来，写出next WORD（移到下个单词）或者redo LINE（重处理当前这一行）之类的代码也很自然。

注19：　这毕竟不是goto语句。

注20：　起码，这么做要比随意命名更有意义。你就算把循环的标签设为XYZZY或是PLUGH，Perl也不会介意。可要是不熟悉20世纪70年代的Colossal Cave游戏，就没人会知道你这么写是什么意思。

条件操作符?:

当Larry考虑Perl要提供哪些操作符时，他不想让老的C程序员有机会怀念那些C有而Perl没有的东西。所以他把C所有的操作符都搬过来了 [注21] 。这个决定导致Perl拥有了C语言中，最让人困惑的操作符，也就是条件操作符?:。虽然它可能令人困惑，不过有时也相当有用。

条件操作符就像将一个if-then-else控制结构。由于使用时需要三个操作数，所以有时也称为三目操作符。它看起来像这样：

```
expression ? if_true_expr : if_false_expr
```

首先，Perl执行条件表达式，看看究竟是真还是假。如果为真，则执行冒号前的表达式；否则，就执行冒号后的表达式。每次使用时都会执行问号右边两个表达式中的一个，另一个则会被跳过。换句话说，若条件表达式为真，则第二个表达式会被求值并返回，而忽略第三个表达式；若条件表达式为假，则忽略第二个表达式，而对第三个表达式求值并返回。

在下面的例子里，子程序&is_weekend的执行结果决定了哪个字符串表达式会被赋值给变量：

```
my $location = &is_weekend($day) ? "home" : "work";
```

下面的例子中，我们会计算并输出一个平均值，或是在无法计算时用一行连字符代替：

```
my $average = $n ? ($total/$n) : "-----";
print "Average: $average\n";
```

任何使用?:操作符的表达式都可以改写成if结构，但常常会更拖沓冗长：

```
my $average;
if ($n) {
    $average = $total / $n;
} else {
    $average = "-----";
}
print "Average: $average\n";
```

这可能是你喜欢的技巧，用来写出干净利落的多路分支：

```
my $size =
    ($width < 10) ? "small"  :
```

注21：　严格说来，他其实舍弃了在Perl中无用的操作符，例如将数字转换成变量的内存地址的操作符。当然他还加上了几个让C程序员嫉妒的操作符，比如字符串连接操作符。

```
($width < 20) ? "medium" :
($width < 50) ? "large"  :
                "extra-large"; # 默认值
```

实际上，这是由三层嵌套的?:操作符组成的。而且你一旦明白其诀窍所在，就会觉得十分简洁。

当然，这个操作符并不是非用不可，初学者可能看了头疼。不过，迟早你会在其他人的程序里看到它，而我们希望有一天你也会在自己的程序里使用它。

逻辑操作符

和你猜想的一样，Perl拥有全套的逻辑操作符，可以用来对付布尔（真/假）值。比如常用来组合逻辑测试的逻辑AND（与）操作符（&&）和逻辑OR（或）操作符（||）：

```
if ($dessert{'cake'} && $dessert{'ice cream'}) {
    # 两个条件都为真
    print "Hooray! Cake and ice cream!\n";
} elsif ($dessert{'cake'} || $dessert{'ice cream'}) {
    # 至少一个条件为真
    print "That's still good...\n";
} else {
    # 两个条件都为假，什么也不干（我们有点伤感）
}
```

Perl在这里可能会走捷径。如果逻辑与操作符的左边表达式为假，整个表达式就不可能为真，因为必须两边都为真才会得到真。因此这时不必再检查右边的表达式，从而避免对其求值。针对下面的例子，请考虑$hour是3的时候会怎样：

```
if ( (9 <= $hour) && ($hour < 17) ) {
    print "Aren't you supposed to be at work...?\n";
}
```

相似的地方还有，若逻辑或操作符的左边表达式为真，那么右边也会免于求值。请考虑下面的例子中$name是fred会如何：

```
if ( ($name eq 'fred') || ($name eq 'barney') ) {
    print "You're my kind of guy!\n";
}
```

因为这样的行为，这种操作符被称为“短路（short-circuit）”逻辑操作符。它们只要有可能就会走捷径来获得结果。事实上，依赖这种短路行为的代码很常见，比如求得平均值的程序：

```
if ( ($n != 0) && ($total/$n < 5) ) {
    print "The average is below five.\n";
}
```

这个例子里，右边的表达式只有在左边为真的时候才会被求值，因此程序不会因为意外地“除以零”而崩溃（我们会在第十七章的“捕获错误”一节展示其他相关的例子）。

短路操作符的值

和C这类语言不同的地方在于，Perl的短路操作符求得的值不只是简单的布尔值，而是最后运算的那部分表达式的值。这提供了一样的结果，因为当整个测试应该为真时，最后部分的值总是真，而当整个测试应该为假时，则最后部分的值总为假。

但是这个返回值会很有用，我们常常利用逻辑或操作符提供变量的默认取值：

```
my $last_name = $last_name{$someone} || '(No last name)';
```

如果`$someone`在哈希中并不存在，左边的计算结果就是`undef`，也就是假。所以逻辑或操作符必须对右边的表达式求值，使它成为左侧变量的默认值。这个习惯用法中，默认值不只是为`undef`准备的，也是为所有逻辑假的值准备的。要进一步细分的话可以使用条件操作符：

```
my $last_name = defined $last_name{$someone} ?
    $last_name{$someone} : '(No last name)';
```

这太麻烦了，而且必须书写`$last_name{$someone}`两遍。好在Perl 5.10提供了更简洁的写法，请看下一节。

定义或操作符

前一节我们谈到||操作符能用于提供变量的默认值。但没有考虑到特殊情况，就是已定义的假值也可能会被意外地替换为默认值。因此后来又改用更丑陋的条件操作符版本。

为了避免这样的问题，Perl 5.10引入了定义或（defined-or）操作符（`//`），在发现左边的值属于已定义时进行短路操作，而不管该值属于逻辑真还是逻辑假。所以就算有人的名字是`0`，代码也能正常工作：

```
use 5.010;

my $last_name = $last_name{$someone} // '(No last name)';
```

有时候需要给一个未定义变量赋值，若已定义则保留原值。比如程序常常会参考`VERBOSE`环境变量来决定是否打印信息。现在可以检查`%ENV`哈希中`VERBOSE`键的值是否定义，若未定义则对它赋值：

```
use 5.010;
```

```
my $Verbose = $ENV{VERBOSE} // 1;
print "I can talk to you!\n" if $Verbose;
```

可以多弄几个值来测试一下//，看看它会在哪种情况下返回default：

```
use 5.010;

foreach my $try ( 0, undef, '0', 1, 25 ) {
    print "Trying [$try] ---> ";
    my $value = $try // 'default';
    say "\tgot [$value]";
    }
```

输出显示只有在$try是undef的时候才会收到default字符串：

```
Trying [0] --->     got [0]
Trying [] --->      got [default]
Trying [0] --->     got [0]
Trying [1] --->     got [1]
Trying [25] --->    got [25]
```

有时候你想要对一个未定义的变量赋值。比如，当你开启警告功能并打印一个未定义的值，就会收到烦人的警告消息：

```
use warnings;

my $name;  # 没有值，未定义！
printf "%s", $name; # 发出警告：Use of uninitialized value in printf ...
```

当然这种错误常常无害，可以忽略。但如果确实要打印未定义的值，则可以用一个空字符串来代替：

```
use 5.010;
use warnings;


my $name;  # 没有值，未定义！
printf "%s", $name // '';
```

使用部分求值操作符的控制结构

之前看到的4个操作符&&、||、//和?:都有一个共性：根据左边的值决定是否计算右边的表达式。有些情况下会执行的表达式在另外的情况下并不执行。因此这些操作符有时也被称为“部分求值（*partial-evaluation*）操作符”。部分求值操作符是天然的控制结构，因为不会执行所有的表达式[注22]。并非Larry热衷于在Perl里加入更多的控制结

注22：　看到这里之前，没准已经有人纳闷，为什么本章要介绍这些逻辑操作符呢？

构，而是因为在他决定将部分求值操作符加进Perl的那一刻，它们就成了天然的控制结构。毕竟任何能激活或停用某段程序代码的东西都算是控制结构。

好在只有当受控制的表达式有副作用时才会引起人们注意到这点，比如修改变量或者输出信息等等，像下面这段代码：

```
($m < $n) && ($m = $n);
```

你应该会立刻注意到，逻辑与的运算结果并没有被赋值到任何地方 [注23] 。为什么呢？

如果$m真的小于$n，左边就为真，所以会执行右边的赋值运算。不过若是$m不小于$n，则左边为假并导致右边被跳过。所以，上面的程序代码基本上和下面这一行做的是同样的事情，但后者显然比较容易理解：

```
if ($m < $n) { $m = $n }
```

或者采用修饰符的写法：

```
$m = $n if $m < $n;
```

你或许会在维护某个程序时看到如下这一行：

```
($m > 10) || print "why is it not greater?\n";
```

如果$m真的大于10，那么左边为真，因此逻辑或运算就算完成了。若非如此，则左边为假，于是会接着输出信息。跟上面一样，这其实可以（而且很可能应该）使用传统的写法，可能是用if或unless来写 [注24] 。

而那些特别思维奇特的人甚至学会了用读英语的方式来读这几行程序代码。比如：检查$m是否小于$n，若是如此，则进行赋值；检查$m是否大于10，若非如此，则输出信息。

会以这种方式来写控制结构的人通常以前是C程序员或是早期的Perl程序员。为什么他们会这样写呢？有些人是因为抱有这样比较有效率的错觉，也有些人认为这些技巧能让他们的程序比较酷，还有一些人只是模仿别人的编程风格而已。

条件操作符同样可以成为控制结构。在下面的例子里，我们想将$x赋值给两个变量中较小的那个：

```
($m < $n) ? ($m = $x) : ($n = $x);
```

注23：　不过别忘了，如果它是子程序的最后一项表达式的话，就会成为返回值。

注24：　大多数这类风格的表达式都是原来编写shell脚本的人按照以往的习惯带过来的。

如果$m比较小，它会得到$x；否则的话，就归$n所有了。

逻辑与和逻辑或操作符还有另一种写法。你可以将它写成单词：and和or [注25] 。这种单词操作符和标点符号形式的效果相同，但是前者在运算优先级上要低得多。既然单词操作符不会紧紧地"粘住"附近的表达式，它们需要的括号可能就会少一些：

```
$m < $n and $m = $n;  # 写成相应的if语句版本会更好
```

当然，可能你还是要用到更多的圆括号，因为优先级是一个怪兽。如果对优先级不是非常有把握，请回来使用圆括号。然而单词操作符的优先级很低，所以你通常可以想象它们会把表达式拆成两片，先做左边所有的事情，如果需要再做右边所有的事情。

尽管以逻辑操作符作为控制结构可能令人困惑，但有时候这却是大家都认可的程序写法。比如下面就是打开文件时的习惯写法 [注26] ：

```
open my $fh, '<', $filename
  or die "Can't open '$filename': $!";
```

通过使用低优先级的短路or操作符，我们向Perl表达了"open this file……or die"的意思。如果文件打开成功，就会返回真，此时or就不必执行了；但如果文件打开失败，or就还得去执行右边的部分，也就是丢出信息并通过die终止程序。

所以，用这些操作符作为控制结构是Perl习惯用语的一部分，也就是Perl约定俗成的用法。适当使用的话，程序会更具威力；否则，你的程序将会难以维护。请别滥用它们 [注27] 。

习题

以下习题答案参见第323页上的"第十章习题解答"一节：

1. [25]编写程序，让用户不断猜测范围从1到100的秘密数字，直到猜中为止。程序应该以神奇公式int(1+rand100) [注28] 来随机产生秘密数字。当用户猜错时，程序应该回应"Too hight"或"Too low"。如果用户键入quit或exit等字样，或是键入一个空白行，程序就应该中止。当然，如果用户猜到了，程序也应该中止！

注25：　其实还有低优先级的not操作符（和逻辑否定操作符 !类似），另外也有很少见的xor异或操作符。

注26：　除非使用autodie编译指令。

注27：　对于以上这些怪异代码（除了or die），一个月写一次以上就可以算是滥用。

注28：　若是好奇，可请参考*perlfunc*文档中关于int和rand函数的介绍。

2. [10]修改前一个练习的程序，打印额外的调试信息，例如程序选择的秘密数字。确保修改的部分可以用开关控制，而且调试开关即使关上也不会产生警告信息。如果在使用Perl 5.10或更新的版本，可以使用`//`操作符，否则请使用条件操作符。

3. [10]修改第六章的习题3中的程序（环境变量列表程序），打印出那些未定义的环境变量（显示为`undefined value`）。可以在程序中设定新的环境变量，来测试程序是否正确打印那些具有假值的变量。如果在使用Perl 5.10或更新的版本，可以使用`//`操作符，否则请使用条件操作符。

第十一章

Perl模块

关于Perl的故事远不止我们在这本书里提到的，有许多人还用它来做各种有趣的事。如果碰上什么棘手的问题难以解决，多半可以在Perl综合典藏网（CPAN）上找到别人现成的解决方案。CPAN在世界各地都有服务器与镜像站点，包含了上千个可用的开源Perl模块。实际上，自从Larry把它设计为可扩展的语言起，绝大部分Perl 5代码都是以模块的形式存在的。

我们这里不打算教你如何编写模块，要学的话可以自己看“羊驼书”。本章我们会教你如何使用现有的模块，目的是让你开始熟悉CPAN，而不是为你评判各个模块的优劣。

寻找模块

Perl模块有两种来源：一种是随Perl发行版本一同打包的，所以安装了Perl就可以用这些模块；另一种则需要从CPAN下载，需要自己安装。除非特别说明，一般我们讨论的都是随Perl一同发布的模块[注1]。

要找寻那些没有随Perl发布的模块，可以到CPAN Search网站（*http://search.cpan.org*）根据模块类型浏览或者直接搜索。在下载完整的模块包之前你可以先阅读详细的模块文档。当然，你也可以在线查看模块包含的所有文件。此外还有许多工具可以提供模块内部的各类信息供你参考。

注1：　某些供应商会为他们自己的Perl发行版提供更多附加的模块。这其实就是所谓的第三方模块，即供应商附送的模块，就像小礼品一样。此外，或许还有一些附加的工具什么的已经安装到你的操作系统中，不妨找找看。

在找寻模块之前，请先检查一下系统上是否已经安装过。可以试着用*perldoc*命令打开模块的文档看看。比如Perl自带的CGI.pm模块（我们稍后会在“CGI模块”一节详细介绍），可以用下面这条命令打开它的文档阅读：

```
$ perldoc CGI
```

改用不存在的模块名称试试看，你会看到一条错误信息。

```
$ perldoc Llamas
No documentation found for "Llamas".
```

你的系统上可能还安装有其他格式的文档，比如HTML格式的。当然，前提是模块本身提供原始文档[注2]。

Perl自带的cpan命令可以创建*autobundle*文件[注3]，它会列出所有你已经安装了的模块，包括版本号：

```
$ cpan -a
```

安装模块

如果想要安装系统上没有的模块，一般来说，需要先下载打包发布的模块文件包，解压缩后在shell中运行一系列编译安装命令。具体步骤和注意事项可以查阅模块包中的*README*文件或*INSTALL*文件。

如果模块使用MakeMaker[注4]封装，可以用下面的流程安装：

```
$ perl Makefile.PL
$ make install
```

如果你没有权限安装模块到系统级目录（安装后其他用户也可使用的目录），则可以在*Makefile.PL*后面加上INSTALL_BASE参数，指定以你的用户身份可写的安装目录：

```
$ perl Makefile.PL INSTALL_BASE=/Users/fred/lib
```

注2：　我们在“羊驼书”中会介绍Perl文档的各方面细节，不过现在，只要记住Perl绝大多数的文档都是写在和源代码相同的那个文件里的。

注3：　所谓的bundle文件，就是CPAN客户端可以用来重新安装当前你已经安装过的那些模块的清单，不管是在同一台机器上，还是在别的新机器上。

注4：　它其实是一个Perl自带的模块ExtUtils::MakeMaker。它提供的一系列工具可以帮你生成模块安装文件，保证其中的安装指令适应Perl和操作系统两方面的环境要求。

有些Perl模块开发者用的是另一个辅助模块Module::Build来编译并安装他们的作品。此时安装流程大致如下：

```
$ perl Build.PL
$ ./Build install
```

和之前一样，你可以指定自己的安装目录：

```
$ perl Build.PL --install_base=/Users/fred/lib
```

有些模块的工作依赖于其他模块，所以必须先安装好这些前置模块，才能继续编译安装。与其自己手动一个个安装模块，不如用Perl自带的CPAN.pm [注5] 。你可以在命令行启动CPAN.pm自己的shell，以便下达相关命令：

```
$ perl -MCPAN -e shell
```

就算这样，写起来还是有些麻烦，所以一段时间之前本书作者之一（brian d foy）写了个小小的脚本程序，叫做*cpan*，它也是Perl自带的并且通常跟*perl*及一些相关的工具安装在一起。现在，只要把想安装的模块的名称列在它后面就行了：

```
$ cpan Module::CoreList LWP CGI::Prototype
```

“我没有命令行！”或许你会这样抱怨。如果你使用的是ActiveState公司发行的Perl版本（有针对Windows、Linux和Solaris的版本），还可以使用Perl包管理器（Perl Package Manager，简称PPM） [注6] 来安装模块。你甚至还可以取得ActiveState出品的CD或DVD [注7] 。除了上面提到的，其他特别的操作系统也应该会有各种安装软件的方式，当然也包括安装Perl模块在内。

另外还有一个非常轻巧的工具*cpanm*（它是*cpanminus*的简写），不过它目前还不是Perl自带的工具。它被设计为零配置、轻量级的CPAN客户端，能完成绝大多数人的日常工作。你可以从*http://xrl.us/cpanm*下载该脚本文件到本地。

注5：扩展名“.pm”表示“Perl Module”，为了与其他概念相区分，通常谈到一些流行的模块时都会带上“.pm”。在这里，“CPAN（Perl综合典藏网）”和“模块CPAN”不是同一个东西，为了有所区别，我们把后者称为“CPAN.pm”。

注6：参见*http://aspn.activestate.com/ASPN/docs/ActivePerl/faq/ActivePerl-faq2.html*。

注7：你可以建立一个本地仓库用以制作你自己的CD或者DVD光盘。尽管现在CPAN有将近4GB大，但你可以用*minicpan*（创意来自本书作者）来下载所有模块的最新版本，而全部这些只要500MB就够了。参见CPAN::Mini模块。

有了*cpanm*脚本后，只要简单地告诉它你要安装的模块名称即可：

```
$ cpanm DBI WWW::Mechanize
```

安装到自己的目录

Perl模块安装过程中最常困扰人们的，是CPAN工具默认会把模块安装到与*perl*解释器相同的目录，但你可能没有往这个系统级别的目录写文件的权限。

对初学者来说，最容易的解决办法是用`local::lib`模块安装新模块到自己的用户目录下。目前这个模块还不是Perl自带的，所以你得自己从CPAN下载安装。这个模块会自动修改某些环境变量设定，借此影响CPAN客户端安装模块的位置。在命令行上加载该模块而不做任何操作，就能列出它所改动的所有环境变量设定[注8]：

```
$ perl -Mlocal::lib
export PERL_LOCAL_LIB_ROOT="/Users/fred/perl5";
export PERL_MB_OPT="--install_base /Users/fred/perl5";
export PERL_MM_OPT="INSTALL_BASE=/Users/fred/perl5";
export PERL5LIB="...";
export PATH="/Users/brian/perl5/bin:$PATH";
```

如果使用-I开关，*cpan*客户端就会参照上面列出的环境变量安装指定的模块[注9]：

```
$ cpan -I Set::Crossproduct
```

相比之下，*cpanm*则要聪明一些。如果你已经设置了那些`local::lib`会帮你设置的环境变量的话，它会直接按照这些设定安装。如果没有，它会检查对默认的安装目录是否拥有可写权限。如果没有写权限，它会自动帮你加载`local::lib`模块。如果你想显式声明使用`local::lib`，只需这样做：

```
$ cpanm --local-lib HTML::Parser
```

稍有经验的用户会配置他们的CPAN客户端，这样以后就一直安装到指定目录。

你可以在CPAN.pm配置中设定下面这些参数，以后使用CPAN.pm shell时就会自动安装新模块到指定的私有目录。为了兼容，你需要配置两个设定，一个给`ExtUtils::Makemaker`用，一个给`Module::Build`用：

注8：　只管照我们给出的命令尝试好了，虽然我们还没介绍过命令行开关方面的知识，不过所有相关的内容都在*perlrun*文档中，请自行查阅。

注9：　你需要最新版的 CPAN.pm，或最新版的App::Cpan模块。现在`local::lib`提供的功能已经整合到 Perl 5.14 里面了。

```
$ cpan
cpan> o conf makepl_arg INSTALL_BASE=/Users/fred/perl5
cpan> o conf mbuild_arg "--install_base /Users/fred/perl5"
cpan> o conf commit
```

注意，这里设定的是和`local::lib`给出的相同的目录。在CPAN.pm配置文件中加上这些设定后，每次安装模块都会自动附加这些安装参数。

在选定安装Perl模块的路径之后，还要告诉应用程序到哪里才能找到这些模块文件。如果用的是`local::lib`，只需简单地在程序内部加载该模块：

```
# 在你的 Perl 程序内部
use local::lib;
```

如果你装在其他地方，可以使用编译指令`lib`指定这个路径：

```
# 也是在你的Perl程序内部
use lib qw( /Users/fred/perl5 );
```

作为起步，上面这些应该够了。我们在《Intermediate Perl》一书中会有更深入的讨论，包括教你编写自己的模块。其他相关信息不妨参阅*perlfaq8*文档。

使用简易模块

假设在程序里有个包含路径的长文件名，比如*/usr/local/bin/perl*，而你希望取出文件的基名（basename）而不包括目录。这其实很简单，文件的基名就是最后一个斜线之后的内容（此例中即为*perl*）：

```
my $name = "/usr/local/bin/perl";
(my $basename = $name) =~ s#.*/##;  # 糟糕!
```

和你之前看到的一样，Perl会先在圆括号中进行赋值，然后再进行替换操作。这里的替换操作将结尾为斜线的字符串（也就是文件路径的目录部分）替换成空字符串，这样就只剩下基名了。或者采用更为简洁的写法，在替换操作中使用`/r`开关：

```
use 5.014;
my $name = "/usr/local/bin/perl";
my $basename = $name =~ s#.*/##r;  # 糟糕!
```

如果这么写，看起来好像可以正常工作。好吧，这只是看起来，实际上这么写隐含着三个问题。

首先，Unix上的文件或目录名称可能会包含换行符（这虽然不是常常发生，但确实可能发生）。由于正则表达式的点号（`.`）无法匹配换行符，像`"/home/fred/flintstone\n/brontosaurus"`这样的文件全名便无法正常运作——此段代码会认为文件的基名是

"flintstone\n/brontosaurus"。你可以用模式的/s选项加以修正（如果你考虑到这种微妙而又罕见的情况的话），写成：s#.*/##s。

第二个问题是，这段代码仅仅考虑了Unix下的情况。它假设目录分隔符总是Unix风格的斜线，而没有考虑其他系统（比如使用反斜线或冒号）的情况。也许你自己写的代码不会拿到那些系统上运作，不过大多数有用（也有些并不怎么有用）的脚本会考虑兼容性，就算拿到罕见系统上也能运作如常。

第三个，也是最大的一个问题是，我们正在试图解决别人早已解决的问题。Perl自带了相当多的标准模块，它们作为Perl的扩展，增加了许多新的特性和功能。若这还不够，CPAN上还有更多好用的模块，每周都有许多新模块加入，每天都有许多老模块更新。如果需要某些模块提供的功能，你（或者是你的系统管理员）可以去那里下载安装，然后直接使用。

我们会在本章后续的章节中选取一些随Perl一同发布的简易模块，演示它们的部分特性和用法（事实上，这些模块还可以做更多的事，眼下只是管中窥豹，大略展示一下如何使用这些简易模块）。

本书无法介绍关于模块使用的全部知识，因为你得先了解引用和对象之类的高级主题才有办法使用某些模块[注10]。这些主题（以及该如何创建模块等等）都在“羊驼书”《Intermediate Perl》里有详细精妙的阐释。而本节则可以帮你从简易模块的使用开始起步。想要进一步了解某些有趣且实用的模块，请阅读附录B。

File::Basename模块

在前面的例子中，我们用了不可移植的方式取得基名。这种解决问题的方式看起来简洁，却可能因为微妙的差异而失败（比如这里就假设文件名或目录名中不会出现换行符）。而且我们还“重新发明了轮子”，解决别人早就解决过（也调试过好几次）的问题。不必感到难为情，其实我们每个人都有这样的坏毛病。

要从文件全名里取出基名，这里有个更好的做法，就是使用Perl自带的File::Basename模块。只要执行perldocFile::Basename命令或通过特定平台的文档系统，你就能阅读它的使用说明。这是使用新模块的第一步（没准也会是第三步与第五步）。

注10：　如同我们在后面几页中将要看到的一样，你可以使用那些采用了对象和引用的模块，但并不需要完全理解这些高级主题。

很快你准备好要使用它，在程序开头的地方用use指令声明加载该模块[注11]：

```
use File::Basename;
```

在程序编译阶段，Perl看到这行代码后，会尝试找寻此模块的源代码并加载进来。接着就好像Perl突然多出了一些新函数，你在程序接下来的部分都可以随意使用这些函数了[注12]。此前的例子中我们需要的正是basename函数：

```
use File::Basename;

my $name = "/usr/local/bin/perl";
my $basename = basename $name;  # 返回'perl'
```

嗯，这样虽然在Unix上行得通，但如果我们的程序在MacPerl、Windows或是VMS等系统上运行呢？不必担心，这个模块会判断你当前用的是哪种操作系统，并且使用该系统默认的文件命名规则（当然，这时$name里存放的就是属于该系统的文件名字符串了）。

此模块还提供了一些其他函数，比如dirname函数可以从文件全名里取得目录名称。这个模块也能让你将文件名和扩展名分开，或是改变默认的文件名规则[注13]。

仅选用模块中的部分函数

如果你想在已有的程序里加上File::Basename模块，却发现该程序中已经有个叫做&dirname的子程序——也就是说，程序里现有的子程序名和模块里的某个函数同名。[注14]。现在麻烦来了，通过使用模块而引入的dirname也是个Perl子程序，该如何与自己写的同名子程序区分开来呢？

只需在File::Basename的use声明中加上导入列表（import list）来指明要导入的函数清单，就不会自动导入所有函数了。在此，我们只需要basename函数：

注11：　一般我们会在文件的头部声明加载这些模块，这样程序维护员比较容易弄清楚到底用了哪些模块。并且，如果你要在新机器上安装程序的话，安装部署也会简单很多。

注12：　你猜到了：事情并非如此简单，有时候得用包名和及其完全限定名称。当你的主程序超过几百行（不算你使用的模块的代码行），成为较大的Perl程序后，你就得学习使用这些高级特性了，请从阅读*perlmod*文档开始学习这方面的知识。

注13：　当你试图处理从Windows机器获得的Unix机器的文件名时（比如通过FTP连接发送命令的时候），你可能需要修改文件名规则。

注14：　好吧，你可能确实有个名为&dirname的子程序，但它做的却是另外一件事，这里只是举个例子。因为某些模块提供数百个新函数（这是真的！），使得命名冲突更加常见。

```
use File::Basename qw/ basename /;
```

如果像底下这么写的话，就表示我们完全不要引入任何新函数：

```
use File::Basename qw/ /;
```

我们也常写成空的括号表示一个都不导入：

```
use File::Basename ();
```

为什么要这么做呢？嗯，这条指令告诉Perl加载File::Basename模块，这和前面一样，但不要导入任何函数名称。导入函数的目的是要使用简短的函数名称，像basename和dirname。然而，哪怕不导入这些名称，我们还是可以通过全名的方式来调用相应的函数：

```
use File::Basename qw/ /;       # 不导入函数名称

my $betty = &dirname($wilma);  # 使用我们自己的子程序&dirname

                                # （略去该子程序的具体内容）

my $name = "/usr/local/bin/perl";
my $dirname = File::Basename::dirname $name;  # 使用模块中的dirname函数
```

如你所见，模块里的dirname函数的全名是File::Basename::dirname。加载模块后，无论是否导入dirname这种简短名称，我们都可以随时使用函数全名的方式来调用。

大多数情况下，使用模块的默认导入列表就行了。不过，你随时可以用自己定义的列表。一来可以略去你不需要的默认导出函数；二来还可以按需要导入那些不会自动导出的函数，因为大多数模块的默认导入列表里都会省略某些（不常使用的）函数。

没错，有些模块默认的导入列表就是特别长。所有模块的说明文档是都应该列出可供导入的符号（symbol），如果有的话，但你随时都可以用自己的列表覆盖掉默认的导入列表，就像上面File::Basename例子里的做法一样。而空列表意味着不导入任何符号。

File::Spec模块

现在我们可以方便地提取文件名的基名了。不光如此，我们还经常需要把基名和目录名结合起来以取得文件全名，比如*/home/rootbeer/ice-2.1.txt*这样的文件全名。为基名加上前缀的做法如下：

```
use File::Basename;

print "Please enter a filename: ";
chomp(my $old_name = <STDIN>);
```

```
my $dirname  = dirname $old_name;
my $basename = basename $old_name;

$basename =~ s/^/not/;  # 给基名加上前缀
my $new_name = "$dirname/$basename";

rename($old_name, $new_name)
    or warn "Can't rename '$old_name' to '$new_name': $!";
```

看出问题在哪了吗？和前面一样，我们假设文件名遵循Unix的惯例，即目录名后接着斜线，再接着文件的基名。幸好，Perl也提供了能够解决这个问题的模块。

你可以利用File::Spec模块来操作文件说明（*file specification*），也就是文件名、目录名以及文件系统里的其他名称。它和File::Basename一样会在运行时判断操作系统的类型，并且总是采用正确规则。但File::Spec是面向对象的（Object Oriented，简称OO）模块，这点和File::Basename不同。

要是你从未因OO热潮而激动，请不必烦心。如果你了解对象是什么，那非常好，你可以使用这个OO模块；要是你不知道何谓对象，那也没关系，只要照样键入我们展示给你的代码，它就会正常工作，时间久了你也就渐渐明白了。

在这个例子里，File::Spec的说明文档告诉我们应该使用catfile这个方法（*method*）。“方法”是什么意思呢？就目前我们所关心的来说，它只不过是另一种函数而已。不同的是，你必须通过File::Spec的全名方式来调用方法，例如：

```
use File::Spec;

.
.  # 取得$dirname和$basename的值，方法同上
.

my $new_name = File::Spec->catfile($dirname, $basename);

rename($old_name, $new_name)
    or warn "Can't rename '$old_name' to '$new_name': $!";
```

如你所见，调用方法时的全名应该由模块的名称（此处称为类（*class*））、一个瘦箭头（->）以及方法的简短名称构成。请注意，这里用的是瘦箭头，而不是File::Basename里的双冒号，看到瘦箭头，就说明是面向对象的写法，后面是要调用的方法名。

不过，既然我们通过全名来调用方法，那么模块会导入哪些符号呢？答案是什么都没有。对OO模块来说，这是正常的做法。这样一来，你就不必担心自己的子程序会跟File::Spec里众多方法重名了。

你会讨厌在程序里使用这些模块吗？其实你总是可以自行决定的。比方说，假设你确定

自己的程序不会在Unix之外的系统上运行，你又完全了解Unix文件名的规则[注15]，那么你也许可以将上述假设写死在程序中。但用这些现成模块总能让你节约不少时间，轻松构造更健壮的程序，而且让程序移植起来也更容易，不另收费。

Path::Class模块

虽然File::Spec模块能很好地处理不同系统上的文件名规则，不过用起来还是有些麻烦的。而Perl自带的Path::Class模块则提供了更为友好的操作界面：

```
my $dir     = dir( qw(Users fred lib) );
my $subdir  = $dir ->subdir( 'perl5' );     # Users/fred/lib/perl5
my $parent  = $dir->parent;                # Users/fred

my $windir  = $dir ->as_foreign( 'Win32' ); # Users\fred\lib
```

CGI.pm模块

如果要创建CGI程序（本书不作此方面介绍），请用CGI.pm模块[注16]。就算你明白甚至精通CGI交互过程中的所有细节，也不必在自己的脚本中亲自实现那些着实让许多人困扰的接口操作和输入信息的解析。CGI.pm的作者Lincoln Stein已花了相当多的时间来确保此模块在大部分服务器和操作系统上都能正常运作。直接使用此模块，然后你就可以腾出精力专注于脚本中真正有趣的部分。

CGI模块有两种风格：古朴的函数接口和面向对象接口。我们将使用前一种。在此之前，你可以照着CGI.pm文档中的例子依样画葫芦。下面的简单CGI脚本会解析CGI输入，并以纯文本的方式来显示输入字段的名称和值。我们在“导入列表”中使用了:all，这是一种导出标签（*export tag*）写法，用来指定一组要导出的函数而非一个，这和之前看到的例子有所不同[注17]：

```
#!/usr/bin/perl

use CGI qw(:all);

print header("text/plain");
```

注15：　如果你不知道文件名和目录名可以包含换行符的话（我们之前提到过的），那就说明你不完全了解这些规则，对吧？

注16：　和CPAN.pm模块类似，谈到它的时候我们会把CGI.pm中的“.pm”读出来，以区别于CGI这个协议。

注17：　该模块还提供了其他导出标签以供选择不同的函数分组。比如说，如果你只想处理CGI的那一组，就可以只导入 :cgi；若要产生HTML的那一组，只导入:html4就可以了。进一步信息请参考CGI.pm文档。

```
foreach $param ( param() ) {
    print "$param: " . param($param) . "\n";
}
```

我们还可以输出HTML格式的结果，看起来就会更花哨些，而CGI.pm有许许多多的实用函数可以派上用场。它可以用start_html()来处理CGI标头，也就是HTML文档开头的部分，另外还有为数众多的和HTML标签同名的函数可供使用，比如用h1()处理H1标签：

```
#!/usr/bin/perl

use CGI qw(:all);

print header(),
    start_html("This is the page title"),
    h1( "Input parameters" );

my $list_items;
foreach my $param ( param() ) {
    $list_items .= li( "$param: " . param($param) );
}

print ul( $list_items );

print end_html();
```

不难吧？你不必知道CGI.pm是怎么办到这一切的，你只需相信它做得到就行了。一旦学会放手让CGI.pm去做所有困难的事情，你便能专注在程序里有趣的部分了。

CGI.pm模块还能做更多的事，像处理cookie信息、页面重定向以及多重页面表单等等。限于篇幅，我们不作详细讨论，要继续学习的话可以参考该模块文档中的例子。

数据库和DBI模块

DBI（Database Interface，数据库接口）模块并未随附于Perl标准发行版本中，但它却是最热门的模块之一，因为许多人或多或少地需要连接数据库。DBI的美妙之处在于，不管哪种常见的数据库，都可以用相同的接口对其进行操作，从CSV文件到Oracle之类的大型数据库服务器。它还具有ODBC驱动程序，而且有些驱动程序是数据库厂商自行提供的。想通盘了解完整细节，请参阅《Programmingthe Perl DBI》一书，由Alligator Descartes和Tim Bunce合著（O'Reilly）。你也可以访问DBI模块的官方网站*http://dbi.perl.org/*。

安装完DBI之后，你还必须安装DBD（Database Driver，数据库驱动程序）。在CPAN上搜索一下就能得到一长串DBD列表。请安装与你的数据库服务器对应的驱动程序，并确

定其版本与你的服务器一致。

DBI是面向对象的模块，但你不必为了要用它而去了解OO编程的全部细节，学习文档中的例子就好了。想要连接数据库，你得用use加载DBI模块并调用它的connect方法：

```
use DBI;

  $dbh = DBI->connect($data_source, $username, $password);
```

变量$data_source内含你要使用的DBD的特定信息，所以你需要从DBD取得。对PostgreSQL数据库来说，驱动程序是DBD::Pg模块，所以$data_source就像这样：

```
my $data_source = "dbi:Pg:dbname=name_of_database";
```

连上数据库之后，就可以进行准备查询、执行查询以及读取查询结果等一系列操作：

```
my $sth = $dbh->prepare("SELECT * FROM foo WHERE bla");
$sth->execute();
my @row_ary  = $sth->fetchrow_array;
$sth->finish;
```

完成工作后，断开与数据库的连接：

```
$dbh->disconnect();
```

当然，还有许多其他DBI能做的事，具体请参阅它的文档。虽然有些内容可能已经过时，不过《Programming the Perl DBI》一书仍然是该模块的优秀指南。

处理日期和时间的模块

能够处理日期和时间的模块有许许多多，不过最为流行的还是Dave Rolsky编写的DateTime模块。它提供所有完整的解决方案，能够处理复杂的时区、日期计算以及其他许多事情。你可以从CPAN获取该模块。

最常见的是把系统中以秒数表示的当前时间（或者epoch纪元时间）转换成DateTime对象：

```
my $dt = DateTime->from_epoch( epoch => time );
```

此后，通过各式对象方法，即可获取该日期的相关信息：

```
printf '%4d%02d%02d', $dt->year, $dt->month, $dt->day;
```

该模块还有格式化输出的时间字符串的方法：

```
print $dt->ymd;            # 2011-04-23
print $dt->ymd('/');       # 2011/04/23
print $dt->ymd('');        # 20010423
```

如果有两个DateTime对象，可以做一些数学计算，看看中间间隔了多久。DateTime是通过重载相应的数学计算操作符实现相关计算的：

```
my $dt1 = DateTime->new(
     year      => 1987,
     month     => 12,
     day       => 18,
     );

my $dt2 = DateTime->new(
     year      => 2011,
     month     => 5,
     day       => 1,
    );

my $duration = $dt2 - $dt1;
```

由于日期计算较为复杂，我们无法简单地把一个间隔时间表示成单个数字：

```
my @units = $duration->in_units( qw(year month day) );

printf '%d years, %d months, and %d days', @units;
```

根据上面定义的两个日期，实际输出的间隔时间为：

```
23 years, 4 months, and 14 days
```

当然我们也可以把一个间隔时间加到某个日期对象上。比如要求得$dt2之后的第5天，可以先创建一个间隔日期对象，然后两者相加：

```
my $duration = DateTime::Duration->new( days => 5 );
my $dt3 = $dt2 + $duration;
print $dt3->ymd;              # 2011-05-06
```

如果只是用到DateTime提供的一小部分功能，则可以换用其他更为轻便的模块。比如，仅仅需要将时间表示为简单的对象，可以用Time::Piece模块，它会重载Perl内置的localtime函数，使其返回一个时间对象，而不再是一个长长的时间要素清单。此外它还提供了许多便捷的函数，用于按照不同方式表示日期部分，比如将月份表示成英文名字，而不仅仅是简单的数字：

```
use Time::Piece;

my $t = localtime;
print 'The month is ' . $t->month . "\n"; # 输出The month is Apr
```

从Perl 5.10起，`Time::Piece`模块就是Perl自带的。如果你在用早期版本，请到CPAN下载安装。

习题

以下习题答案参见第325页上的“第十一章习题解答”一节。记住，你需要从CPAN下载安装某些模块，某些习题还要求通过查阅CPAN上的文档来熟悉某些模块的使用：

1. [15]从CPAN安装`Module::CoreList`模块。输出Perl5.14自带的所有模块的清单。请建立一个哈希，其键为指定Perl版本自带模块的名称，可以使用下面这行代码：

```
my %modules = %{ $Module::CoreList::version{5.014} };
```

2. [20]写一个程序，用`DateTime`模块计算当前日期和输入日期之间的间隔。输入日期时，在命令行依此键入表示年月日的数字：

```
$ perl duration.pl 1960 9 30
50 years, 8 months, and 20 days
```

第十二章

文件测试

在此之前，我们已经演示过如何打开文件并输出文件内容。通常，打开文件的操作会直接创建一个新文件，如果存在同名文件的话，还会清空该文件的内容。有些时候，你可能需要先检查一下同名文件是否存在；有些时候，你可能需要看看给定的文件究竟存在多久；或者有些时候，你可能需要比较一些文件的大小，看是否都在某个级别以上并且是否已经有一段时间无人使用。对于这类信息的取得，Perl有一套完整的文件测试操作符可供使用。

文件测试操作符

Perl提供了一组用于测试文件的操作符，借此返回特定的文件信息。所有这些测试操作符都写作-X的形式，其中X表示特定的测试操作（实际上还有一个字面写作-X的文件测试操作符，所以泛指和特指常常容易让人混淆）。绝大多数测试操作符返回布尔真假值。虽然称它们为操作符，但实际上它们相应的文档却是写在*perlfunc*里面[注1]。

在运行那些会创建新文件的程序前，应先检查指定文件是否已经存在，以免意外覆盖重要电子表格或是宝贵生日档案。要达到此目的，我们可以用-e文件测试操作符来测试文件是否存在：

```
die "Oops! A file called '$filename' already exists.\n"
  if -e $filename;
```

注1：　要查看完整清单，可以通过命令行执行 *perldoc -f -X*。这里的-X就是字面上的，并非命令行开关。它表示显示所有文件测试操作符相关的文档，*perldoc*无法单个列出其中某个测试符。

请注意，这个例子中die抛出的消息中并没有包含$!，因为我们现在并不关心系统为何拒绝文件访问的请求。接下来的例子是要测试某个文件是否能保证持续更新。我们测试的是一个已经存在的文件句柄，而非表示文件名的字符串。假设我们需要某个程序的配置文件保持每周或每两周更新一次（也许这是一个病毒资料库）。如果文件在过去28天里都没变动过，显然是出了问题。这里的-M文件测试操作符返回的是文件最后一次修改时间到当前程序启动时刻之间的天数，说起来有点饶舌，看看实际代码就明白了，用起来还是很方便的：

```
warn "Config file is looking pretty old!\n"
  if -M CONFIG > 28;
```

第三个例子就复杂多了。假设我们的硬盘空间已满，但是不想花钱再买硬盘，于是我们决定找出最大而且很久没用到的文件，将它们移到备份磁带上。我们需要遍历文件列表[注2]，看看哪些是大于100KB的文件。在确定某个文件够大之后，我们还得确定它已经超过90天没被访问过（这样我们才确信它不太常用），才可以将它移到备份磁带。这里的-s文件测试操作符返回的并不是布尔真假值，而是以字节计算的文件大小（已经存在的空文件大小可以是零字节）[注3]：

```
my @original_files = qw/ fred barney betty wilma pebbles dino bamm-bamm /;
my @big_old_files;  # 将要移到备份磁带上的既大且旧的文件列表
foreach my $filename (@original_files) {
  push @big_old_files, $filename
    if -s $filename > 100_000 and -A $filename > 90;
}
```

所有这些文件测试符看起来都是同一种形式：连字符加上一个字母，字母表示测试的意义，后面跟上要测试的文件名或文件句柄。大多数文件测试操作符返回的都是布尔真假值，少数返回的是表示特别意义的数据。要学习了解其他特例，请参阅表12-1中的完整清单并仔细阅读后面关于特例的说明。

表12-1：文件测试操作符及其意义

文件测试操作符	意义
-r	文件或目录，对目前（有效的）用户或组来说是可读的
-w	文件或目录，对目前（有效的）用户或组来说是可写的

注2：　实际上并不会像此例那样在数组中放置一连串的文件。一般来说，会使用“文件名通配（glob）”或是“目录句柄（directory handle）”来从文件系统中读取。因为这里还没提到目录句柄（参见第十三章），我们只好以列表来示意了。

注3：　还有种比这个例子效率更高的实现方法，你会在本章末看到。

表12-1：文件测试操作符及其意义（续）

文件测试操作符	意义
-x	文件或目录，对目前（有效的）用户或组来说是可执行的
-o	文件或目录，由目前（有效的）用户拥有
-R	文件或目录，对实际的用户或组来说是可读的
-W	文件或目录，对实际的用户或组来说是可写的
-X	文件或目录，对实际的用户或组来说是可执行的
-O	文件或目录，由实际的用户拥有
-e	文件或目录，是存在的
-z	文件存在而且没有内容（对目录来说永远为假）
-s	文件或目录存在而且有内容（返回值是以字节为单位的文件大小）
-f	是普通文件
-d	是目录
-l	是符号链接
-S	是socket类型的文件
-p	是命名管道，也就是先入先出（fifo）队列
-b	是块设备文件（比如某个可挂载的磁盘）
-c	是字符设备文件（比如某个I/O设备）
-u	文件或目录设置了setuid位
-g	文件或目录设置了setgid位
-k	文件或目录设置了sticky位
-t	文件句柄是 TTY 设备（类似系统函数 isatty() 的测试；不能对文件名进行此测试）
-T	看起来像文本文件
-B	看起来像二进制文件
-M	最后一次被修改后至今的天数
-A	最后一次被访问后至今的天数
-C	最后一次文件节点编号（inode）被变更后至今的天数

-r、-w、-x和-o这几个操作符测试的是有效用户或组的ID [注4]，看它们是否有相应的文

注4：　测试操作符-o及-O只管用户ID，而不理会组ID。

件权限。所谓有效用户，指的是“负责”运行这个程序的人[注5]。这些测试会查看文件的“权限位（permission bit）”，以此判断哪些操作是允许的。如果系统使用访问控制列表（Access Control List，简称ACL），那么测试将根据该列表进行判断。上述测试只能返回系统对操作的看法，但受实际情况限制，允许的事未必真的可行。比如说，对某个放在CD-ROM中的文件做-w测试可能为真，可实际上你却无法修改此文件；而对某个空文件进行-x测试时也可能会返回真，但实际上空文件又怎么执行呢？

如果文件内容不为空，-s会返回表示文件大小的数字，单位是字节。若仅用于条件判断，非零数字亦代表真值。

Unix文件系统上[注6]有且仅有7种文件类型，分别可由以下7种文件测试操作符代表：-f、-d、-l、-S、-p、-b和-c。任何条目都应该符合其中一种。但如果你有指向某个文件的符号链接，那么-f和-l都会为真。所以，如果你想要知道某个文件是否为符号链接，最好先测试-l（我们将会在第十三章的“链接与文件”学到更多关于符号链接的知识）。

文件时间测试操作符，-M、-A和-C（是的，都是大写），分别会返回从该文件最后一次被修改、被访问或者它的inode被更改后到现在的天数[注7]（inode是文件系统的索引条目，其中记录了某个文件的所有属性信息，但文件内容除外，相关细节请参阅系统函数stat的文档，或找本详细介绍Unix内部细节的书看看）。天数的值用浮点数表示，如果是两天零一秒之前被修改的文件，可能会返回像2.00001这样的值（这里所说的天数并不是平常所说的自然天。举例来说，假如你在凌晨1:30的时候检查某个曾在午夜前一小时修改过的文件，则-M的返回值大约是0.1，也就是过去的小时数换算到天的数字，这是一个相对时间）。

在检查文件的时间记录时，可能会得到像-1.2这样的负数，这表示文件最后一次被访问的时间戳是在未来30小时后！实际上，程序开始运行的那一刻会被记录下来作为检查时

注5：好学的人请注意：相应的-R、-W、-X及-O测试使用的是实际用户或组的ID。这在你的程序以set-ID方式运行时相当重要。在这种情况下，它是调用程序的用户的ID。请参阅详细介绍高级Unix程序设计的书，进一步了解set-ID程序的概念。

注6：这在许多非Unix文件系统下也行，但不是每种文件测试在任何地方都有意义。比如非Unix系统上大概就没有块设备文件。

注7：在非Unix系统上可能会有所不同，因为有些系统记录的时间与Unix不同。例如在某些系统上，ctime字段（即-C测试操作符所查询的字段）是文件的创建时间（Unix并不跟踪这个时间），而不是inode改变的时间。请参阅*perlport*文档。

间的原点[注8]，所以负值可能表示已经运行很久的程序找到某个刚刚才被访问过的文件。或者是谁不小心（也可能是故意的）将文件属性设成未来时间的。

至于-T和-B，则会测试某个文件是文本文件还是二进制文件。但是对文件系统有一点经验的人都知道，（至少在与Unix类似的操作系统下）没有任何位会告诉你它是二进制文件还是文本文件——那么Perl是如何办到的呢？答案是Perl会作弊：它先打开文件，检查开头的几千个字节，然后作出一个合理的猜测。如果它看到很多空字节、不寻常的控制字符而且还设定了高位（即第八位是1）的字节，那么它看起来就是二进制文件；如果文件里没有许多奇怪的东西，而且它看起来像文本文件，那就猜测为文本文件。如你所料，这样总会有猜错的时候。所以这种猜测并不完美，不过如果你只想把编译过的文件和源文件分开，或是将HTML文件和PNG文件分开，那么这两种测试操作符还算够用。

你可能会以为-T和-B出现的结果必定相反，因为文件若不是文本文件，就该是二进制文件。但是，有两种特殊情况会让测试结果相同：如果文件不存在，两者都会返回假，因为它既不是文本文件也不是二进制文件；在空文件的情况下，两者都会返回真，因为它既是空的文本文件也是空的二进制文件。

关于-t的测试，如果被测试的文件句柄是一个TTY设备，测试的返回值就为真——简单来说，如果该文件可以交互，就判断为TTY设备，所以普通文件或管道（pipe）都可以排除在外。当-tSTDIN返回真的时候，通常意味着可以用交互方式向用户提出一些问题；若返回值为假，那就表示输入来源是个普通文件或是管道，而不是键盘[注9]。

至于其他文件测试操作符，如果你不知道它们的意义也不用担心——要是你没听说过，多半也不太会用到。但如果你感兴趣的话，找一本详细介绍Unix程序设计的书吧。（在非Unix系统中，这些测试都会尽力模仿Unix系统的实现，要是碰上某个无法实现的功能，就会返回undef。通常你都可以猜到实际测试的结果。）

如果文件测试操作符后面没写文件名或文件句柄（也就是只写了-r或-s），那么默认的操作数就是$_里的文件名。文件测试操作符-t是个例外，因为此项测试对文件名（不可能是TTY）而言无用武之地，所以默认情况下它测试STDIN。所以，若要测试一连串的文件名来找出哪些是可读的话，可以这么做：

```
foreach (@lots_of_filenames) {
  print "$_ is readable\n" if -r;  # 亦即 -r $_
```

注8：　这取决于$^T变量的值。如果想取得相对于不同起始时刻的天数，可以改变它的值（可用$^T = time;这样的语句）。

注9：　也许采用IO::Interactive模块加以判断才是更好的选择，因为实际情况可能非常繁杂多变。该模块文档中对它能处理的各种情形有专门的阐述。

```
}
```

但如果省略参数，请特别小心，任何接在文件测试操作符之后的，就算看起来不像，也会被当作要测试的目标。比方说，你想知道以KB为单位（而不是以字节为单位）的文件大小，那么你可能会直接把-s操作的结果除以1000（或是1024），像这样：

```
# 文件名放在$_里
my $size_in_K = -s / 1000;  # 糟糕!
```

当Perl的语法解析器看到斜线时，不会认为那是一个除法操作符。因为它在看到-s操作符后，就要尝试寻找它的可选操作数，也就是要测试的文件，现在它看到的是一个斜线开头的，像是某个正则表达式，但却找不到结尾，于是报错。要避免这种问题，可在文件测试操作符的两边加上括号：

```
my $size_in_k = (-s) / 1024;  # 默认使用$_作测试
```

当然，明确写上要测试的文件总是安全稳妥的做法。

测试同一文件的多项属性

如果要一次测试某个文件的若干属性，可以将各个文件测试组成一个逻辑表达式。比如我们只想操作那些既可读又可写的文件，可以依次检查这两个属性并用and合并起来：

```
if (-r $file and -w $file) {
  ... }
```

这可是个非常耗费资源的操作，每次进行文件测试时，Perl都得从文件系统中取出所有相关信息（实际上，每次Perl都在内部做了一次stat操作，下节我们就会介绍）。虽然在做-r测试的时候，我们已经拿到了所有相关信息，可到了做-w测试的时候，Perl又要去取一遍相同的信息。多浪费啊！如果是对海量文件测试各种属性，性能问题就会非常明显。

Perl有个特别的简写可以避免这种重复劳动。它就是虚拟文件句柄_（就是那个下划线字符），它会告诉Perl用上次查询过的文件信息来做当前测试。现在，Perl只需要查询一次文件信息即可：

```
if (-r $file and -w _) {
    ... }
```

我们并非只能在一条语句中连续使用_。以下就是在两条if语句中使用的例子：

```
if (-r $file) {
  print "The file is readable!\n";
}
```

```
if (-w _) {
  print "The file is writable!\n";
}
```

这么用的时候，必须要清楚代码最后一次查询的是否为同一个文件。若是在两个文件测试之间又调用了某个子程序，那么最后一个查询的文件可能会变化。比如说，下面的例子调用lookup子程序，在其内部对另一个文件做了一次测试。当子程序返回后，再执行文件测试时，文件句柄_代表的就不是原先的$file，而是$other_file：

```
if (-r $file) {
  print "The file is readable!\n";
}

lookup( $other_file );

if (-w _) {
  print "The file is writable!\n";
}

sub lookup {
  return -w $_[0];
}
```

栈式文件测试操作符

在Perl 5.10之前，如果要一次测试多个文件属性，只能分开为若干独立的操作，哪怕是为了节省点力气用虚拟句柄_也不例外。比如说，我们想要测试某个文件是否可读写，就必须分别做可读测试和可写测试：

```
if (-r $file and -w _) {
  print "The file is both readable and writable!\n";
}
```

若能一次完成就好了。从Perl 5.10开始允许我们使用“栈式（stack）”写法将文件测试操作符排成一行，放在要测试的文件名前：

```
use 5.010;

if (-w -r $file) {
  print "The file is both readable and writable!\n";
}
```

这个使用栈式写法的例子和上一个例子做相同的事，仅是语法上有所改变。注意，使用栈式写法时，靠近文件名的测试会先执行，次序为从右往左。不过通常测试次序不是很重要。

对于复杂情况来说，这种栈式文件测试特别好用。比如我们想要列出所有可读、可写、可执行并隶属于当前用户的所有目录，只需要按恰当的顺序摆上这些测试操作符：

```
use 5.010;

if (-r -w -x -o -d $file) {
  print "My directory is readable, writable, and executable!\n";
}
```

对于返回真假值以外的测试来说，栈式写法并不出色。像下面的例子，我们原本想要确认某个小于512字节的目录，可实际上会出问题：

```
use 5.010;

if (-s -d $file < 512) {    # 错啦！千万不要这么做
  say 'The directory is less than 512 bytes!';
}
```

按其内部的实现方式展开，我们可以看到上面的例子实际上相当于如下的写法，整个合并起来的文件测试表达式成了比较运算的一个操作数：

```
if (( -d $file and -s _ ) < 512) {
  print "The directory is less than 512 bytes!\n";
}
```

当-d返回假时，Perl将假值同数字512作比较。比较的结果就变为真，因为假等效为数字0，而0永远小于512。为了避免这种令人困惑的写法，还是用分开的方式写比较好，这对将来维护程序的人来说也更友善：

```
if (-d $file and -s _ < 512) {
  print "The directory is less than 512 bytes!\n";
}
```

stat和lstat函数

用前面介绍的文件测试操作符已经可以获取某个文件或文件句柄的各种常用属性，但这只是一部分，还有许多其他属性信息没有对应的文件测试操作符。比如说，没有任何文件测试操作符会返回到文件的链接的个数或是该文件拥有者的ID（uid）。想知道文件所有其他相关信息，请使用stat函数。此函数能返回和同名的Unix系统调用stat近乎一样

丰富的文件信息（总之比你想知道的还多）[注10]。函数stat的参数可以是文件句柄（包括虚拟文件句柄_），或是某个会返回文件名的表达式。如果stat函数执行失败（通常是因为无效的文件名或是文件不存在），它会返回空列表；要不然就返回一个含13个元素的数字列表，具体意义见下面由标量变量构成的列表：

```
my($dev, $ino, $mode, $nlink, $uid, $gid, $rdev,
  $size, $atime, $mtime, $ctime, $blksize, $blocks)
    = stat($filename);
```

这些变量名代表stat函数返回的列表中对应的数据，你应该抽出点时间看看stat(2)文档里的详细说明。不过在这里，我们会简单列出比较重要的几个：

$dev和$ino

即文件所在设备的编号与文件的inode编号。这两个编号决定了这个文件的唯一性，就像发给文件的“牌照（license plate）”一样。即使它具有多个不同的文件名（使用硬链接创建），设备编号和inode编号的组合依然是独一无二的。

$mode

文件的权限位集合，还包含其他信息位。如果你曾用Unix命令*ls-l*查看过详细（冗长）的文件列表，你会看到其中每一行都是由类似-rwxr-xr-x这样的字符串开始的。长度共9个字符的连字符和字母，刚好对应到$mode最低有效的9个权限位[注11]。以这个例子而言，其实就是八进制数值0755。而其他的位则表示文件的其他信息。所以如果需要用到权限模式，那你就得学会如何使用按位运算操作符（本章稍后就会介绍）。

$nlink

文件或目录的（硬）链接数，也就是这个条目有多少个真实名称。这个数值对目录来说总会是2或更大的数字，对文件来说则（通常）是1。我们会在第十三章的“链

注10：在非Unix系统上，stat、lstat以及文件测试操作符都应该返回“可能范围内最接近的值”。比如，没有用户ID的系统（也就是以Unix观点来看，该系统刚好只有一个“用户”）可能会返回零，以表示用户及组ID，将唯一的用户当成系统管理员。如果stat或lstat执行失败，便会返回空列表。如果文件测试操作符的底层系统调用失败（或者在该系统上无法取得），则该测试操作符通常会返回undef。请参阅*perlport*文档里各种系统上预期结果的最新数据。

注11：字符串里的第一个字符并不代表权限位，而是表示此条目的类型：其中连字符代表一般文件，d代表目录，l代表符号链接。ls命令是根据最低九位之上的其他位来判断文件类型。

接与文件”一节介绍如何为文件创建链接，那时可以了解更多有关链接的信息。在ls-l的结果中，权限位之后的数字就是文件链接数。

$uid和$gid

以数值形式表示文件拥有者的用户ID及组ID。

$size

以字节为单位的文件大小，和-s文件测试操作符的返回值相同。

$atime，$mtime和$ctime

三种时间戳，但在这里是以系统的时间格式来表示：一个32位的整数，表示从纪元（*epoch*）算起的秒数，纪元是测量系统时间的任意基准点。在Unix与某些其他的系统中，纪元是从公元1970年世界标准时间的午夜算起，但也有些系统上会不同。本章稍后将会进一步阐述如何将时间戳的值转换成有用数据。

对符号链接名调用stat函数将会返回符号链接指向对象的信息，而非符号链接本身的信息（除非该链接指向的对象目前无法访问）。若你需要符号链接本身的信息（多半没用），你可以用lstat（它会返回同样顺序、同样意义的内容）来代替stat。如果lstat的参数不是符号链接，它会和stat一样返回空列表。

就像文件测试操作符一样，stat和lstat的默认操作数是$_。也就是说，底层的stat系统调用会对标量变量$_里的文件名进行操作。

localtime函数

你能获得的时间戳值（比如从stat函数返回的时间戳）看起来通常会像1180630098这样的形式。对大多数人来说，这样的数字实在不太方便，除非你想通过减法来比较两个时间戳的大小。所以，你可能需要将它转换成比较容易阅读的形式，比如“Thu May 31 09:48:18 2007”这样的字符串。Perl可以在标量上下文中使用localtime函数完成这种转换：

```
my $timestamp = 1180630098;
my $date = localtime $timestamp;
```

在列表上下文中，localtime会返回一个数字元素组成的列表，但其中有些元素并不是你想要的：

```
my($sec, $min, $hour, $day, $mon, $year, $wday, $yday, $isdst)
  = localtime $timestamp;
```

$mon是范围从0到11的月份值，很适合用来索引月份名。$year是一个自1900年起算的

年数，所以将这个值加上1900就是实际的年份。$wday的范围从0（星期天）到6（星期六），$yday则表示目前是今年的第几天，范围从0（1月1日）到364或365（12月31日）。

还有两个相关的函数对你可能也有用。gmtime函数和localtime函数一样，只不过它返回的是世界标准时间（即格林威治标准时间）。如果需要从系统时钟取得当前的时间戳，可以使用time函数。不提供参数的情况下，不论localtime或gmtime函数，默认情况下都使用当前time返回的时间值：

```
my $now = gmtime;  # 取得当前世界标准时间的时间戳字符串
```

关于操作时间及日期的进一步信息，请参考附录B所提到的诸多有用模块。

按位运算操作符

如果你需要逐位进行运算，比如对stat函数返回的权限位进行处理，就必须用到按位运算操作符（bitwise operator）。该操作符对数据执行二进制数学运算。按位与操作符（*bitwise-and*，记作&）会给出两边参数对应的位置中哪些位同时都为1。举个例子，表达式10&12得到的值是8。按位与操作符只有在两边相应的位均为1的状况下才会产生1。因此，10（写成二进制是1010）和12（二进制为1100）的按位与运算结果是8（二进制1000，也就是10和12以二进制数字表示时同时为1的位所组成的数字）。请参考图12-1。

```
  1010
&1100
  ----
  1000
```

图12-1：按位与操作

表12-2列出了各种按位运算操作符及其意义。

表12-2：按位运算操作符及其意义

表达式	意义
10 & 12	按位与——哪些位在两边同时为真（此例得8）
10 | 12	按位或——哪些位在任一边为真（此例得14）
10 ^ 12	按位异或——哪些位在任何一边为真，但不能两边都为真（此例得6）

表12-2：按位运算操作符及其意义（续）

表达式	意义
6 << 2	按位左移——将左边操作数向左移动数位，移动位数由右边操作数指定，并以0来填补最低有效位（此例得24）
25 >> 2	按位右移——将左边操作数向右移动数位，移动位数由右边操作数指定，并丢弃移出的最低有效位（此例得6）
~10	按位取反，也称为取反码——返回操作数逐位反相之后的数值（此例得0xFFFFFFF5，但请参考后文的说明）

好吧，下面来看几个例子，看看我们可以用这些操作符对stat函数返回的$mode信息如何进行位操作。位操作的结果可以给chmod使用（我们会在第十三章中介绍此函数）：

```
# $mode是从配置文件 CONFIG 的stat信息中取出的状态值
warn "Hey, the configuration file is world-writable!\n"
  if $mode & 0002;                              # 配置文件有安全隐患
my $classical_mode = 0777 & $mode;              # 遮蔽额外的高位
my $u_plus_x = $classical_mode | 0100;          # 将一个位设为1
my $go_minus_r = $classical_mode & (~ 0044);    # 将两个位都设为0
```

使用位字符串

所有这些按位运算操作符既可以操作位字符串（bitstring），也可以对整数进行操作。如果操作数都是整数，结果也会是整数（整数最少会是一个32位的整数，但是如果你的硬件支持更多位的整数，它也可能会更大。例如在64位的机器上，~10的结果会是0xFFFFFFFFFFFFFFF5，而不是32位机器上的0xFFFFFFF5。）

不过，假如按位运算操作符的任一操作数是字符串，则Perl会把它当成位字符串来处理。换句话说，"\xAA"|"\x55"的结果会是"\xFF"。注意，这个例子里的值都是单字节（single-byte）的字符串，而结果是8位都为1的字节。Perl对位字符串的长度没有限制。

这是少数Perl区分字符串和数字的地方。如果想了解利用按位运算操作符处理位字符串的细节，请参阅*perlop*文档。

习题

以下习题答案参见第326页上的“第十二章习题解答”一节：

1. [15]写一个程序，从命令行取得一串文件名，并汇报这些文件是否可读、可写、可执行以及是否确实存在。（提示：如果你可以写一个函数一次做完这些测试会很方便。）如果先对文件做*chmod*为0，你的程序会汇报什么？（也就是说，如果你

使用Unix系统，chmod 0 *some_file*这样的命令就会把文件标示成不可读、不可写也不可执行。）在大部分的shell下，用星号作为参数，代表当前目录下的所有文件。也就是说，你可以用./ex12-2*这样的命令来向程序一次询问多个要测试文件的属性。

2. [10]写一个程序，从命令行参数指定的文件中找出最旧的文件并且以天数汇报它已存在了多久。若列表是空的（也就是命令行中没有提及任何文件），那么它该做什么？

3. [10]写一个程序，用栈式文件测试操作符列出命令行参数指定的所有文件，看看拥有者是否是你自己，以及它们是否可读、可写。

第十三章

目录操作

我们在上一章所创建的文件通常保存在和程序文件相同的目录，这么做有点乱。不过现代的操作系统都使用目录来组织和管理文件，比如将披头士（Beatles）的MP3文件和我们的“小骆驼书”各章的重要源文件分开存放，以免不小心把MP3文件寄给出版社。Perl让你可以直接对目录进行操作，即使在不同的操作系统下，做法也还是差不多的。

在目录树中移动

程序运行时会以自己的工作目录（*working directory*）作为相对路径的起点。也就是说，当我们提及fred这个文件时，其实指的是“当前工作目录下的fred”。

你可以用chdir操作符来改变当前的工作目录。它和Unix shell的*cd*命令差不多：

```
chdir '/etc' or die "cannot chdir to /etc: $!";
```

因为这是一个对操作系统的调用，所以发生错误时便会设定标量变量$!的值。如果chdir的返回值为假，则表示有些事情没有顺利完成。这时，你应该检查一下$!中的错误原因。

由Perl程序启动的所有进程都会继承Perl程序的工作目录（我们会在第十四章谈到）。可是工作目录的更改却无法影响调用Perl程序的进程比如shell。这并不是Perl本身的限制，这种限制实际上是Unix、Windows和其他系统的一个特性。如果你真的想改变shell的工作目录，请参阅shell的相关文档。这意味着你没办法写出可以代替shell里的cd命令的Perl程序，因为一旦退出Perl程序，又会回到启动Perl程序时所在的工作目录。

如果调用chdir时不加参数，Perl会猜想你要回到自己的用户主目录（home directory）并试着将工作目录设成主目录，这和在shell下使用不加参数的*cd*命令的效果相同。这是少数不以$_作为默认参数的情形之一。

有些shell允许你使用波浪号作为*cd*的前缀，让你能够以另一个用户的主目录作为起点（像*cd~merlyn*）。但这是shell提供的功能，而非操作系统。因为Perl的chdir是通过直接调用操作系统实现的，跳开了shell，所以这里无法使用这种以波浪号开头的写法[注1]。

文件名通配

一般来说，shell会将命令行里的文件名模式展开成所有匹配的文件名。这就称为文件名通配（*glob*）。比如把*.pm这个文件名模式交给*echo*命令，shell会将它展开成名称相匹配的文件列表：

```
$ echo *.pm
barney.pm dino.pm fred.pm wilma.pm
$
```

这里的*echo*命令其实并不知道如何展开*.pm，因为shell会先把*.pm展开成一些符合条件的文件名，然后再交给它处理。这对Perl程序来说也是一样的。下面的程序只是简单输出所有命令行参数：

```
foreach $arg (@ARGV) {
  print "one arg is $arg\n";
}
```

运行程序时，如果只有一个带有文件名通配的参数，shell会先展开该通配模式，再把结果传递给程序。这样，对程序来说，就好比是看到多个参数：

```
$ perl show-args *.pm
one arg is barney.pm
one arg is dino.pm
one arg is fred.pm
one arg is wilma.pm
```

请注意，show-args完全不必了解如何进行文件名通配处理——放在@ARGV里的已经是展开好了的名称。

不过有时候在程序内部也可能需要用*.pm之类的模式。我们可以不花太多力气就把它展开成相匹配的文件名吗？当然！只要用glob操作符就行了：

注1：　你可以试试File::HomeDir模块，不管在什么操作系统上都能进入指定用户的主目录。

```
my @all_files = glob '*';
my @pm_files = glob '*.pm';
```

其中，@all_files会取得当前目录中的所有文件并按字母顺序排序，但不包括以点号开头的文件，这和shell中的做法完全相同。@pm_files得到的列表与之前在令行使用*.pm时的相同。

其实，任何能够在命令行上键入的模式都可以作为（唯一的）参数交给glob处理，如果要一次匹配多种模式，可以在参数中用空格隔开各个模式：

```
my @all_files_including_dot = glob '.* *';
```

其中，我们加上了“.*”参数以取得所有的文件名，无论它们是否以点号开头。请注意，在引号括住的字符串里，两个条目之间的空格是有意义的：它分隔了两个要进行文件名通配处理的条目[注2]。glob操作符的效果之所以和shell完全相同，是因为在Perl 5.6版之前它只不过是在后台调用*/bin/csh*[注3]来展开文件名。因此文件名通配非常耗时，而且还可能在目录太大时（或别的情况下）崩溃。有责任心的Perl黑客会避开文件名通配处理，而改用目录句柄（directory handle），本章稍后会详细讨论。不过，如果你用的是新版的Perl，就不必再担心这件事了。

文件名通配的另一种语法

虽然我们一直在介绍文件名通配，也介绍了glob操作符的用法，可在许多进行文件名通配处理的程序里你可能完全看不到glob这个词。为什么呢？嗯，那是因为大部分过去写的程序都是在glob操作符出现之前写的。它们使用尖括号语法来调用此功能，看起来就跟读取自文件句柄差不多：

```
my @all_files = <*>;    # 效果和这样的写法完全一致：my @all_files = glob "*";
```

Perl会把尖括号内出现的变量替换成它的值，类似于双引号内字符串的变量内插，这表示在进行文件名通配之前，尖括号内的Perl变量会先被展开成它们的当前值：

```
my $dir = '/etc';
my @dir_files = <$dir/* $dir/.*>;
```

此处，因为$dir会被展开成它当前的值，所以最终会取得指定目录下的所有文件，不管名称是以点号开头的文件，还是不以点号开头的。

注2：　Windows用户可能习惯使用*.*这个文件名通配来代表“所有文件”。但其实它表示的是“所有名称中包含点号的文件”，即使对Windows平台上的Perl来说也是这个意思。

注3：　如果C-shell不存在，它会调用其他合适的shell程序。

这样说来，假如尖括号既表示从文件句柄读取又代表文件名通配操作，那Perl又是如何判断取舍的呢？嗯，因为合理的文件句柄必须是严格意义上的Perl标识符，所以如果尖括号内是满足Perl标识符条件的，就作为文件句柄来读取；否则，它代表的就是文件名通配操作。例如：

```
my @files = <FRED/*>;   # 文件名通配操作
my @lines = <FRED>;     # 从文件句柄读取
my @lines = <$fred>;    # 从文件句柄读取
my $name = 'FRED';
my @files = <$name/*>;  # 文件名通配操作
```

上述规则的唯一例外，就是当尖括号内仅是一个简单的标量变量（不是哈希或数组元素）时，那么它就是间接文件句柄读取（*indirect filehandle read*）[注4]，其中变量的值就是待读取的文件句柄名称：

```
my $name = 'FRED';
my @lines = <$name>; # 对句柄FRED进行间接文件句柄读取
```

Perl会在编译阶段决定它是文件名通配操作还是从文件句柄读取，因此和变量的内容无关。

假如你喜欢，也可以使用`readline`操作符来执行间接文件句柄读取[注5]，让程序读起来更清楚些：

```
my $name = 'FRED';
my @lines = readline FRED;  # 从FRED读取
my @lines = readline $name; # 从FRED读取
```

不过，因为间接文件句柄读取并不常见，并且通常也只用在简单的标量变量上，所以很少有用到`readline`操作符的机会。

目录句柄

若想从目录里取得文件名列表，还可以使用目录句柄（*directory handle*）。目录句柄看起来像文件句柄，使用起来也没多大差别。你可以打开它（以`opendir`代替`open`），读取它的内容（以`readdir`代替`readline`），然后将它关闭（以`closedir`代替`close`）。只不过读到的是目录里的文件名（或其他东西的名称），而不是文件的内容。例如：

注4：　如果间接文件句柄是一个文本字符串，那么在`use strict`的情况下将它交给“符号引用（symbolic reference）”来测试是不允许的。不过，间接句柄还可能是符号表通配（typeglob）或某个I/O对象的引用，这样尽管在使用了`use strict`的时候，也是可以正常工作的。

注5：　如果你使用的是Perl 5.005或之后的版本。

```
my $dir_to_process = '/etc';
opendir my $dh, $dir_to_process or die "Cannot open $dir_to_process: $!";
foreach $file (readdir $dh) {
  print "one file in $dir_to_process is $file\n";
}
closedir $dh;
```

和文件句柄一样，目录句柄会在程序结束时自动关闭，也会在用这个句柄再打开另一个目录前自动关闭。

你也可以选用裸字作为目录句柄的名称，这和文件句柄一样，但也同样存在问题：

```
opendir DIR, $dir_to_process
    or die "Cannot open $dir_to_process: $!";
foreach $file (readdir DIR) {
  print "one file in $dir_to_process is $file\n";
}
closedir DIR;
```

这里的实现和文件名通配完全不同，在旧版Perl里面通配操作需要开启多个外部进程，而目录句柄的读取是不需要额外打开什么进程的。所以，对于需要榨取更多计算能力的程序来说，前者可以提供更佳的性能。不过，目录句柄是个低级操作符，所以我们必须自己多做点事。

比如目录句柄返回的名称列表并未按照任何特定的顺序排列[注6]。此外，列表里将会包含所有的文件，而不只是匹配某些模式的部分（好比刚才的文件名通配范例中的`*.pm`）。另外列表里包含了名称以点号开头的文件，要特别注意`.`和`..`也在其中[注7]。因此，如果我们只想处理名称以*pm*结尾的文件，则可以在循环内使用一个跳过函数：

```
while ($name = readdir $dh) {
  next unless $name =~ /\.pm$/;
  ... 其他对文件名的处理 ...
}
```

请注意，这是正则表达式的语法，而不是文件名通配。若想取得所有名称不以点号开头的文件，我们可以这么写：

```
next if $name =~ /^\./;
```

或者，如果要排除`.`（当前目录）和`..`（上层目录）这两个条目，可以直接写明：

注6：　这其实是目录列表里（未经排序）的条目顺序，就像执行*ls -f*或*find*时得到的顺序一样。

注7：　许多古老的Unix程序都犯了这个错误，以为`.`和`..`一定是前两个返回的条目（无论是否排序）。如果你根本没这么想过，请忘记这段脚注，因为这是错误的猜想。事实上，我们现在已经后悔提到它了。

```
next if $name eq '.' or $name eq '..';
```

接下来要说明的是最让人迷惑的地方，所以请特别注意。readdir操作符返回的文件名并不含路径名，它们只是目录里的文件名而已。所以，我们不会看到*/etc/passwd*，而只会见到*passwd*（因为这是另一个与文件名通配操作的区别，所以很容易把人搞糊涂）。

所以得加上路径名称才有办法得到文件的全名：

```
opendir my $somedir, $dirname or die "Cannot open $dirname: $!";
while (my $name = readdir $somedir) {
  next if $name =~ /^\./;            # 跳过名称以点号开头的文件
  $name = "$dirname/$name";          # 拼合为完整的路径
  next unless -f $name and -r $name; # 只需要可读的文件
  ...
}
```

为了让程序更具可移植性，可以用File::Spec::Functions模块构造用于本地系统的合适文件名[注8]：

```
use File::Spec::Functions;

opendir my $somedir, $dirname or die "Cannot open $dirname: $!";
while (my $name = readdir $somedir) {
  next if $name =~ /^\./;             # 跳过点号开头的文件
  $name = catfile( $dirname, $name ); # 拼合为完整的路径
  next unless -f $name and -r $name;  # 只需要可读的文件
  ...
}
```

若是没有接上路径，文件测试操作符会在当前目录下查找文件，而不是在$dirname指定的目录下。这是使用目录句柄时最常犯的错误。

递归访问目录

在你刚开始Perl编程生涯的数十小时内，一般不会需要进行递归目录访问。我们不希望现在就介绍如何用Perl实现*find*这样的命令行工具，以免分心，目前你只需要知道，Perl自带了一个很棒的模块File::Find，我们可以用它漂亮简洁地完成目录的递归处理。我们这么说的目的是希望你不要自己重新写一遍同样功能的子程序，好多初学者在开始的数十小时里都摩拳擦掌地想要自己实现点什么，但过不了多久就会茫然不知所措，对“本地目录句柄”和“如何返回原来的目录”感到困惑。

如果你已经习惯于用Unix的*find*命令完成任务，那你也可以把这些任务改成使用Perl自带

注8：　另外还有一个模块Path::Class，它能完成同样的任务，而且用户界面更为友好，只不过它不是Perl默认自带的模块。

的*find2perl*脚本生成的Perl程序来完成。把原来用在*find*命令上的参数照搬过来，就会自动生成一段同样功能的Perl程序代码：

```
$ find2perl . -name '*.pm'
#! /usr/bin/perl -w
    eval 'exec /usr/bin/perl -S $0 ${1+"$@"}'
        if 0; #$running_under_some_shell

use strict;
use File::Find ();

# Set the variable $File::Find::dont_use_nlink if you're using AFS,
# since AFS cheats.

# for the convenience of &wanted calls, including -eval statements:
use vars qw/*name *dir *prune/;
*name   = *File::Find::name;
*dir    = *File::Find::dir;
*prune  = *File::Find::prune;

sub wanted;

# Traverse desired filesystems
File::Find::find({wanted => \&wanted}, '.');
exit;

sub wanted {
    /^.*\.pm\z/s
    && print("$name\n");
}
```

对于更为复杂的任务，可以搜索下CPAN上的模块，比如`File::Find::Rule`和`File::Finder`。这两个模块都是架构在`File::Find`基础之上，但提供了更为直观、易用的用户界面。

文件和目录的操作

常常有人使用Perl来打理文件和目录，因为Perl是在Unix环境下成长起来，而且仍然主要为Unix服务，所以这一章可能看起来会比较偏向Unix。值得庆幸的是在非Unix系统上，Perl也能以同样的方式工作。

删除文件

大多数时候，我们创建文件是为了让数据有个地方落脚。一旦数据过时，我们应该及时删除文件。在Unix shell下，我们可以键入*rm*命令来删除单个或多个文件：

```
$ rm slate bedrock lava
```

在Perl里面，我们可以使用unlink操作符，并指定要删除的文件列表：

```
unlink 'slate', 'bedrock', 'lava';

unlink qw(slate bedrock lava);
```

这会把三个文件放进碎纸机，从此消失在系统中。

既然unlink的参数是一个列表，glob函数又恰好返回的是一个列表，我们只要联合两者，就可以一次删除多个文件：

```
unlink glob '*.o';
```

这跟我们在shell中的rm *.o很像，但却不必启动外部的rm进程。所以可以更快地让这些重要文件消失！

unlink的返回值代表成功删除的文件数目。所以在第一个例子里，改用下面的写法可以检查它是否执行成功：

```
my $successful = unlink "slate", "bedrock", "lava";
print "I deleted $successful file(s) just now\n";
```

是的，如果返回值是3，我们当然知道所有文件都被删除了；如果是0，则表示没有删除任何文件。那如果是1或2呢？嗯，我们没办法知道被删除的是哪个。假如你一定要知道，请使用循环每次删除一个：

```
foreach my $file (qw(slate bedrock lava)) {
  unlink $file or warn "failed on $file: $!\n";
}
```

因为每次只删除一个文件，所以返回值不是0（失败）就是1（成功），这刚好可以当成布尔值来控制warn的执行。在这里or warn和之前在第五章中介绍的or die的用法相似，只是后果没那么严重。我们在警告信息后面加上了换行符，这样就不会显示是哪行程序代码发出的警告，因为即便文件删除失败，也肯定不是我们程序代码上的问题。

当unlink执行失败时，内置的$!变量会被设成操作系统错误的相关信息。此变量只有在循环处理每一个文件的过程中才可用，因为每次系统调用失败时都会重设该变量的内容。unlink不能用来删除目录（这同不带参数的rm命令不能删除目录一样），你得使用稍后提到的rmdir函数。

在Unix上有个鲜为人知的事实：某个文件可能让你无法读取、写入、执行，甚至无法拥有——其实它根本就是别人的文件，但你还是可以将它删除。这是因为删除文件的权限跟文件本身的权限位无关，它取决于文件所在目录的权限位。

之所以提到这点，是因为Perl的初学者在测试unlink时，常会在建立一个文件后将它chmod成0（这样就无法对它进行读写），看看这是否能让unlink执行失败。可是结果恰好相反，文件却像肥皂泡一样消失了[注9]。不过，如果你真的想看到unlink执行失败，只要试着删除*/etc/passwd*或类似的系统文件就行了。因为这个文件是由系统管理员控制的，因此你无法将它删除[注10]。

重命名文件

想为现有的文件取个新名字？可以用rename函数：

```
rename 'old', 'new';
```

跟Unix的mv命令一样，这会将名为*old*的文件改为同一个目录下名为*new*的文件。你甚至可以将文件移到其他目录中：

```
rename 'over_there/some/place/some_file', 'some_file';
```

有些人喜欢用第六章（“胖箭头”一节）里提到的胖箭头表示改名的先后：

```
rename 'over_there/some/place/some_file' => 'some_file';
```

只要运行程序的用户拥有足够的权限，这会将其他目录中名为some_file的文件移动到当前目录里[注10]。和大部分调用操作系统功能的函数一样，rename执行失败时返回假，并且会将操作系统返回的错误信息存到$!里，从而让你可以（通常也应该）用or die（或是or warn）来向用户汇报问题。

新闻组里最常见[注11]的Unix shell问题就是：如何批量把名称以*.old*结尾的文件改名为以*.new*结尾。下面是Perl拿手的做法：

```
foreach my $file (glob "*.old") {
  my $newfile = $file;
```

注9：　当然，如果你在尝试这种操作时太粗心，忘记了现在正以系统管理员的身份登录，那就没话好说了。

注10：　此外，它们还必须在同一个文件系统里。这条规则存在的理由稍后介绍。

注11：　批量改名不仅是历史常见问题，也是这些新闻组目前最常见的问题，因此也是常见问题集（FAQ）里名列榜首并且最早被解答的问题。就算这样，它还是继续留在榜首。呃……

```
    $newfile =~ s/\.old$/.new/;
    if (-e $newfile) {
      warn "can't rename $file to $newfile: $newfile exists\n";
    } elsif (rename $file => $newfile) {
      # 改名成功，什么都不需要做
    } else {
      warn "rename $file to $newfile failed: $!\n";
    }
  }
```

此程序会先检查$newfile是否存在，因为只要用户具有删除目标文件的权限，rename就会高高兴兴地覆盖掉它。加上这项检查，就可以降低损失数据的几率。当然，如果你原本就是打算要覆盖掉旧文件，比如*wilma.new*，就不必在程序里先用-e来测试了。

循环里的前两行程序代码可以（通常也会）合并成这样：

```
(my $newfile = $file) =~ s/\.old$/.new/;
```

这种做法会先声明$newline并从$file里取得它的初始值，然后对$newfile进行替换。你可以把它读成：用右边的模式将$file变换成$newfile。而考虑到优先级的因素，括号是必需的。

这在Perl 5.14里面借助/r标志的话，可以直接在s///替换时生成新文件名。形式上差不太多，但省掉了括号：

```
use 5.014;

my $newfile = $file =~ s/\.old$/.new/r;
```

没准有些程序员会注意到：怎么替换运算中左边的点号之前有反斜线，而右边却没有。其实这是因为两边的意义不同，左边的部分是正则表达式，右边的则可视为双引号内的字符串。所以我们需要用模式/\.old$/来表示“字符串结尾的.old”（因为不想替换掉*betty.old.old*这个文件名里第一次出现的.old，所以得将锚位定在字符串结尾），但是在右边可以直接写成.new以作为替换字符串。

链接与文件

要进一步了解文件和目录的运作，先搞清楚Unix的文件及目录模型会有所帮助，即使你的系统跟Unix的运作方式稍有差异。限于篇幅，想深入了解文件系统的读者请参考详细讲解Unix内部细节的资料。

“挂载卷（*mounted volume*）”指的是硬盘或相似设备，例如磁盘分区、软盘、CD-ROM或DVD-ROM。其中可能含有任意数量的文件和目录。每个文件都存储在编号的

*inode*对应的位置中，我们可以把它想象成磁盘上的门牌号码。某个文件或许会存在inode 613中，而另一个则可能存在inode 7033中。

不过，寻找某个特定的文件时，我们得从它的目录找起。目录是一种由系统管理的特殊文件。基本上目录是一份文件名和相应inode编号的对照表[注12]。目录列出来的内容当中一定会有两个特殊条目：一个是.（称作“点”），代表目录本身；另一个则是..（称作“点点”），指的是目录结构中的高一层目录（也就是它的上层目录）[注13]。图13-1展示了两个inode：一个是名为*chicken*的文件，另一个是Barney的诗歌目录*/home/barney/poems*，其中包含了*chicken*文件。文件的inode编号是613，而目录的inode编号是919（该目录的名称*poems*并没有在这张图中，因为当前目录本身的信息是被存放在别的目录中的）。目录中有三个文件条目（包括*chicken*）和两个子目录（其中一个inode 919指向当前目录本身）条目以及每个条目的inode编号。

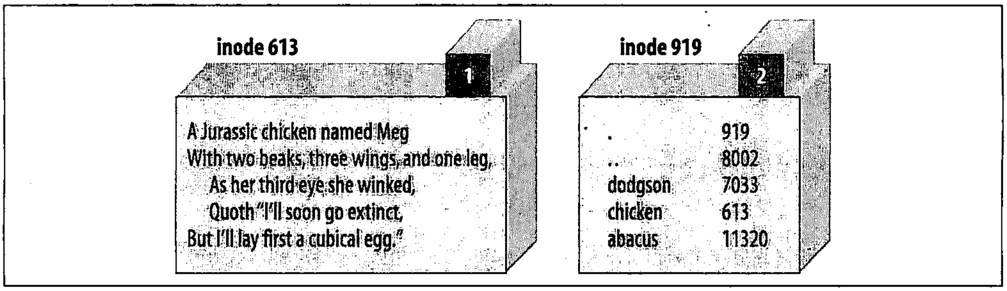


图13-1：先有chicken再有egg

要在指定目录中创建一个新文件时，系统会新增一个条目来记录文件名与新的inode编号。系统怎么知道哪个inode可用呢？答案是每个inode都有自己的链接数（*link count*）。如果inode并未在任何目录里出现，它的链接数就一定是零。因此，所有链接数为零的inode都可以用来存放新的文件。每当inode被列入目录中，链接数就会递增；当它在目录的列表里被删除时，链接数就会递减。对上图里的文件*chicken*来说，inode的链接数是1，我们把链接数显示在inode数据右上方的小框里。

不过，有些inode会出现在多个目录的列表里。举例来说，前面提过每个目录都会有.这个条目，它会指回目录本身的inode。所以任何目录的链接数都至少是2：一个位于它的上层目录的列表里，另一个位于它本身的列表里。除此之外，如果里面有子目录，则每

注12：　在Unix系统上（别的系统通常没有inode、硬链接等概念），可以利用*ls*命令的*-i*选项来查看文件的inode编号。不妨试着键入像*ls -ail*这样的命令。假如文件系统里有两个以上的条目具有相同的inode编号，那么它们实际上是一个文件，只占用一份磁盘空间。

注13：　Unix的根目录（*root*）没有上层目录。在根目录下的..和.一样，都指向根目录本身。

个子目录还会通过..条目再增加一个链接[注14]。在图13-1中，目录的inode链接数为2（显示在右上角的小框里）。链接数代表的是该inode的真实名称的数量[注15]。那么一般文件的inode也可以在目录的列表里重复出现吗？当然可以。假设Barney在上述的目录里用Perl的link函数建立了一个新的链接：

```
link 'chicken', 'egg'
  or warn "can't link chicken to egg: $!";
```

这和在Unix shell命令行上键入ln chicken egg的效果类似。link在成功时会返回真，失败时则会返回假，并且设定$!的值，Barney可以在错误信息里检查这个值。这个程序运行后，*egg*这个名称就会指向文件*chicken*，反之亦然。两个名称现在没有哪个比另一个"真实"，而且（你大概猜到了）要仔细调查才能知道是先有鸡还是先有蛋。图13-2展示了新的情况，图中有两个指向inode 613的链接。

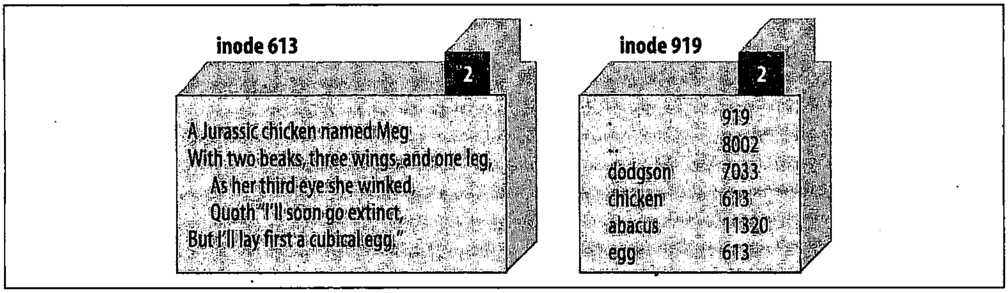


图13-2：egg被链接到chicken

所以，这两个文件名都会指向磁盘上的同一处。如果文件*chicken*里面有200个字节的数据，那么*egg*里也会有相同的200个字节，而且两个加起来总共还是200个字节（因为这只不过是同一份文件的两个名称）。如果Barney为*egg*文件新增了一行文字，则*chicken*文件的结尾处也会出现相同的一行文字[注16]。现在如果Barney不小心（或故意）删除了*chichen*文件，文件里的数据并不会丢失，因为还可以用*egg*这个文件名来访问。反过来

注14：这表示目录的链接数一定等于子目录的数量加上2。有些系统上确实如此，但其他的系统可能有所不同。

注15：在传统的*ls -l*输出中，硬链接的数量会显示在权限标志（如-rwxr-xr-x）的右边。这个数值对目录来说总是大于1，而对普通文件来说则几乎全是1，其中的道理你现在应该知道了。

注16：当你尝试建立链接、更改文件时，请注意大部分的文本编辑器都不会"就地"（in place）编辑文件，而是将改动过的内容存入副本再存回去。如果Barney使用文本编辑器来修改*egg*，那么最后很可能会有一个名为*egg*的新文件以及一个名为*chicken*的旧文件。它们是两个不同的文件，而不是对同一个文件的两个链接。

说如果他删除的是*egg*文件，那么还可以访问*chicken*。当然如果他把两个文件都删了，数据就丢失了[注17]。关于目录列表里的链接还有一条规定：在目录列表中所有inode指向的文件都必须在同一个挂载卷中[注18]。这样一来，即使将物理介质（可能是磁盘）移到另一台机器上，其中的目录和文件间的链接仍然有效。正因为如此，`rename`虽然可以将文件移到别的目录里，但是来源和目的地必须位于同一个文件系统（挂载卷）上。如果要跨磁盘移动文件，就必须重新部署inode的数据。对于简单的系统调用来说，这种操作实在太复杂了。

链接的另一个限制就是不能为目录建立额外的名称。这是因为目录必须按照层次排列，如果没有这条规则，*find*和*pwd*之类的工具程序很快就会在文件系统丛林中迷失了。

因此，不能增加目录的链接数，也不能跨挂载卷链接。幸好，链接的这些固有限制是可以绕过的，只要使用另一种链接方式即可，那就是符号链接（*symbolic link*）[注19]。符号链接（也叫做软链接（*soft link*），以便和前面所说的真正的硬链接（*hard link*）区分开来）是目录里的一种特殊条目，用来告诉系统实际文件放置在别的地方。假设Barney（在那个诗歌的目录下）使用Perl的`symlink`函数建立一个软链接，如下所示：

```
symlink 'dodgson', 'carroll'
  or warn "can't symlink dodgson to carroll: $!";
```

这和Barney在shell下执行*ln -s dodgson*命令的效果相同。图13-3显示了执行的结果，包括inode 7033里的那首诗在内。

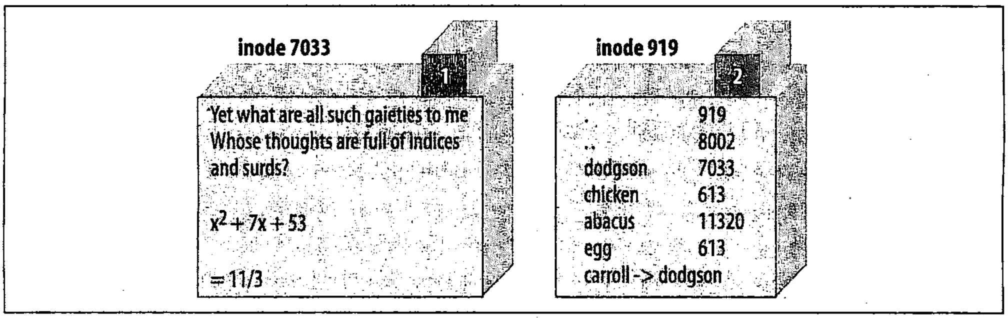


图13-3： 指向inode7033的符号链接

现在如果Barney想读取*/home/barney/poems/carroll*，由于系统会自动跟随符号链接，所

注17： 虽然系统不一定会立刻覆盖掉这个inode，但在链接数减到零时通常很难救回数据。你最近做过备份吗？

注18： 有个例外，就是存储设备根目录里的 .. 条目，它会指向被挂载设备的目录

注19： 有些非常古老的Unix系统不支持符号链接，但近年来这类系统已经非常罕见了。

以结果和直接打开*/home/barney/poems/dodgson*相同。但是这个新名称并不是文件真正的名称，因为（如图13-3所示）inode 7033的链接数仍然只是1而已。符号链接只是告诉系统："你如果是来这里找*carroll*的话，请到dodgson那里去。"

符号链接和硬链接不同，它可以跨文件系统为目录建立软链接（也就是一个新的目录名）。事实上，符号链接能指向任何文件名，而不管它放在哪个目录里，甚至还可以指向不存在的文件！不过，这也表示软链接不像硬链接那样可以防止数据丢失，因为它并不会增加inode的链接数。如果Barney删掉了*dodgson*，系统就不能跟随*carroll*这个软链接了[注20]。虽然*carroll*这个条目还在，但是尝试读取它会得到像`file not found`这样的错误。此时，以`-l 'carroll'`进行文件测试会返回真，而`-e 'carroll'`则会返回假：它是个符号链接，可实际上目标文件并不存在。

由于软链接可以指向目前还不存在的文件，所以在创建文件时也很有用。Barney将大部分文件放在自己的主目录*/home/barney*下，不过他也经常需要访问某个名称很长、难以键入的目录*/usr/local/opt/system/httpd/root-dev/users/staging/barney/cgi-bin*。因此他建立了*/home/barney/my_stuff*这个符号链接来指向那个名称很长的上述目录，这样他要进去就很容易了。假如他（从自己的主目录）创建了*my_stuff/bowling*这个文件，该文件的真实名称将会是*/usr/local/opt/system/httpd/root-dev/users/staging/barney/cgi-bin/bowling*。下个星期，要是系统管理员将Barney的文件移到*/usr/local/opt/internal/httpd/www-dev/users/staging/barney/cgi-bin*目录，那么Barney只要更改符号链接指向的目录，他和他的程序就又可以轻松地找到文件了。

在你的系统上，*/usr/bin/perl*或*/usr/local/bin/perl*（也可能两者皆是）常会是符号链接，都指向真正的Perl二进制文件。假设你是系统管理员，刚编译了一份新版Perl。旧版Perl当然还在运行，而你也不想打乱任何事情。当你准备要更新Perl时，只要更改一两个符号链接就行了。这样一来，任何以*#!/usr/bin/perl*开头的程序都会自动改用新版的Perl。要是出了什么问题（虽然不大可能），只要改回原来的符号链接，就又能切到旧版Perl了。（不过因为你是个好管理员，所以会在改版之前先通知用户测试新的*/usr/bin/perl-7.2*。改版之后，你也会保留旧版的Perl一个月，并让所有需要这段过渡时间的用户将程序的第一行改成*#!/usr/bin/perl-6.1*。）

软硬链接都很有用，这个事实可能让人惊讶。很多非Unix系统没有任何链接机制，因此使用起来非常不方便。可以阅读*perlport*文档，看看这些平台的最新进展如何。

要取得符号链接指向的位置，请使用`readlink`函数。它会返回符号链接指向的位置，如

注20：　当然，删除*carroll*只会删除符号链接而已。

果参数不是符号链接，则返回undef：

```
my $where = readlink 'carroll';              # 得到 "dodgson"

my $perl = readlink '/usr/local/bin/perl';  # 告诉你实际的perl程序究竟躲在何处
```

这两种链接都可以用unlink移除，你现在该了解它取这个名字的含义了吧。unlink只是从目录里移除该文件名的链接条目，并将它的链接数递减，必要时再释放inode。

创建和删除目录

要在现有目录下创建新目录是件很容易的事，只需调用mkdir函数即可：

```
mkdir 'fred', 0755 or warn "Cannot make fred directory: $!";
```

没错，返回值为真表示成功，失败时则会设定$!的值。

可第二个参数0755是什么意思呢？它代表目录建立时的初始权限[注21]（将来随时可以再更改）。写成八进制数值，是因为它会被解释成3位一组的Unix权限值，适合用八进制来表达。没错，就算在Windows或MacPerl上，你也需要略懂Unix的权限值，才有办法使用mkdir函数。0755是个不错的设定，因为它赋予你完整的权限，而其他人只能读取却不能更改任何内容。

注意mkdir函数并不要求你用八进制写这个值，它只是需要某个数字（直接量或运算结果都行）。但除非你能快速心算出八进制的0755等于十进制的493，否则还是让Perl来算比较方便。此外，如果你不小心遗漏了数字开头的零，就会得到十进制的755，这等于八进制的1363，那是一个相当奇怪的权限组合。

正如第二章提过的，想当成数字来用的字符串即使以0开头，也不会被解释成八进制数字。所以下面这么写是行不通的：

```
my $name = "fred";
my $permissions = "0755";  # 危险……不能这么用
mkdir $name, $permissions;
```

糟糕，因为0755会被当成十进制处理，所以相当于我们用奇怪的01363权限值建立了一个目录。要解决这个问题，请使用oct()函数，它能强行把字符串当成八进制数字处理，无论它是否以0开头：

```
mkdir $name, oct($permissions);
```

注21：　权限值通常会被umask的值修改。更详细的说明请参阅umask(2)文档。

当然，在程序中直接指定权限时，不必使用字符串，直接用数字就行了。通常是在用户键入权限值时才会需要额外的oct()函数。举例来说，假设我们从命令行取得参数：

```
my ($name, $perm) = @ARGV;  # 从命令行最先传入的两个参数分别是目录名称和权限
mkdir $name, oct($perm) or die "cannot create $name: $!";
```

变量$perm的值一开始就被当成字符串处理，所以oct()函数会将它正确地解释成通用的八进制表示法。

想移除空目录可以用rmdir函数它的用法和unlink函数很像，只是每次调用只能删除一个目录：

```
foreach my $dir (qw(fred barney betty)) {
  rmdir $dir or warn "cannot rmdir $dir: $!\n";
}
```

如果对非空目录调用rmdir函数会导致失败。你可以先用unlink删除目录中的内容，再试着移除已经清空的目录。举例来说，假设我们需要一个目录来存放程序运行时产生的许多临时文件 [注22]：

```
my $temp_dir = "/tmp/scratch_$$";       # 由进程标识符决定，请参考正文的说明
mkdir $temp_dir, 0700 or die "cannot create $temp_dir: $!";
...
# 将临时目录$temp_dir作为所有临时文件存放的场所
...
unlink glob "$temp_dir/* $temp_dir/.*"; # 删除临时目录$temp_dir中所有的文件
rmdir $temp_dir;                        # 现在是空目录，可以删除了
```

初始的临时目录名将会包含当前的进程标识符，每个当前运行着的进程都有这么一个独一无二的数字代号，在Perl里会把这个代号存储在变量$$中（类似于shell）。这么做是为了避免和别的进程冲突，只要它们也在路径名称里包含进程标识符。（事实上，通常在进程标识符之外还会加上程序名称。所以，如果这个程序名是quarry，那么目录名称多半应该像/tmp/quarry_$$这样。）

在程序结尾处，最后的unlink应该会移除临时目录里所有的文件，然后rmdir函数才有办法将清空后的目录删除。不过，如果我们在临时目录里创建了子目录，那么unlink操作符在处理它们时将会失败，rmdir也会跟着失败。请参考Perl自带的File::Path模块，里面的rmtree函数提供了比较完整的解决方案。

注22：　如果你确实想要创建一个临时文件或目录，可以用Perl自带的File::Temp模块。

修改权限

Unix的*chmod*命令可用来修改文件或目录的权限。Perl里对应的chmod函数也能进行同样的操作：

```
chmod 0755, 'fred', 'barney';
```

和许多其他操作系统接口函数一样，chmod会返回成功更改的条目数量，哪怕只有一个参数，它也会在失败时将$!设成合理的错误信息。第一个参数代表Unix的权限值（即使在非Unix的Perl版本里也一样）。这个值通常会写成八进制形式，理由和前面介绍mkdir时相同。

Unix的*chmod*命令能接受用符号表示的权限（例如+x或go=u-w），但是chmod函数并不接受这类参数[注23]。

修改隶属关系

只要操作系统允许，你可以用chown函数修改一系列文件的拥有者以及其所属组。拥有者和所属组会被同时更改，并且在指定时必须给出数字形式的用户标识符及组标识符。例如：

```
my $user  = 1004;
my $group = 100;
chown $user, $group, glob '*.o';
```

如果要处理的不是数字，而是像merlyn这样的字符串呢？答案很简单，只要用getpwnam函数将用户名转换成用户编号，再用相应的getgrnam[注24]函数把用户组名转换成组编号：

```
defined(my $user = getpwnam 'merlyn') or die 'bad user';
defined(my $group = getgrnam 'users') or die 'bad group';
chown $user, $group, glob '/home/merlyn/*';
```

这里我们用defined函数来检查返回值不是undef，如果指定用户或组不存在就会返回undef。

成功操作后，chown函数会返回受影响的文件数量，在错误发生时会在$!中设定出错信息。

注23：　除非你安装了来自从CPAN的File::chmod模块，这个模块能使chmod操作符升级，从而支持符号表示的权限值。

注24：　这两个函数的名称可说是有史以来最丑陋的。但请不要骂Larry，他只是沿用伯克利那群人取的名字而已。

修改时间戳

在某些罕见情况下，可能需要修改某个文件最近的更改或访问时间以欺瞒其他程序，我们可以用utime函数来造假。它的前两个参数是新的访问时间和更改时间，其余参数就是要修改时间戳的文件名列表。时间格式采用的是内部时间戳的格式（也就是第十二章的“stat和lstat函数”一节中介绍的stat函数的返回值类型）。

“right now”是个很方便的时间戳值，而time函数正是返回这种格式的值的函数。所以，如果想修改当前目录下所有的文件，让它们看起来是在一天前改动过，而最后一次访问就是现在，只要这么写：

```
my $now = time;
my $ago = $now - 24 * 60 * 60;  # 一天的秒数
utime $now, $ago, glob '*';     # 将最后访问时间改为当前时间，最后修改时间改为一天前
```

当然，你可以随意修改文件的时间戳，将它设成未来或过去的任何时间（直到我们能用64位的时间戳之前，在Unix上只能设成1970年到2038年之间，其他系统则可能各有不同）。也许你可以利用这个技巧来创建某个目录，用来存放你写的时间旅行小说的手稿。

在文件有任何更动时，第三个时间戳（ctime值）一定会被设成“now（当前时刻）”，所以utime函数没法修改它（就算用utime修改成功，它也会立刻被设回当前时刻）。这是因为ctime主要是给增量备份的程序用的：如果某个文件的ctime比磁带上的新，那它就该被再次备份了。

习题

下面的程序可能会造成危险！请在没什么重要文件的目录下测试，以免不小心删除重要数据。

以下习题答案参见第329页上的“第十三章习题解答”一节：

1. [12]写一个程序，让用户键入一个目录名称并从当前目录切换过去。如果用户键入一行空白符，则以用户主目录作为默认目录，所以应当会切换到他本人的主目录中。然后输出该目录的内容（不含名称以点号开头的文件）并按照英文字母顺序排列。（提示：用目录句柄还是用文件名通配更容易呢？）如果切换目录失败则应显示警告信息，但不必输出目录内容。

2. [4]修改前题程序，让它输出所有文件，包括名称以点号开头的文件。

3. [5]如果你在前题使用的是目录句柄，那么请以文件名通配重写一次；如果使用的

是文件名通配，那么请以目录句柄重写一次。

4. [6]编写功能和*rm*类似的程序，删除命令行指定的任何文件（不用支持*rm*的所有参数）。

5. [10]编写功能和*mv*类似的程序，将命令行的第一个参数重命名为第二个参数（不必实现*mv*的各种选项或任何额外的参数）。别忘了第二个参数可以是目录。假如它是目录，请在新目录中使用原来的基名。

6. [7]如果你的系统支持，写一个功能和*ln*类似的程序，建立从第一个参数到第二个参数的硬链接（不必实现*ln*的各种选项或额外参数）。如果系统不支持硬链接，那只要输出关于它本来会进行的操作的信息就行了。提示：这个程序和前一题有点像，希望这个提醒可以节省你写程序的时间。

7. [7]如果操作系统支持，请修改上题程序，让它接受可能出现在其他参数之前的`-s`选项。此选项表示要建立的是软链接，而非硬链接（即使系统无法使用硬链接，也请用这个程序试试看是否至少能建立软链接）.。

8. [7]如果操作系统支持，写一个程序，让它在当前目录下查找所有符号链接并输出它们的值（和*ls-l*的格式一样：`name->value`）。

第十四章

字符串与排序

在Perl擅长处理的问题中，约有90%与文本处理有关，其余10%则覆盖了其他领域。所以毋庸置疑，Perl的文本处理能力很强，之前我们用正则表达式解决的那些问题就是明证。不过，有时正则表达式引擎对你而言可能太过花哨，多数时候你只需要简单的字符串处理功能。本章我们就来围绕这个主题谈一谈。

用index查找子字符串

查找子字符串其实也就是要找出它在主字符串中的相对位置。如果是在某个较长的主字符串中，可以借助index函数解决这个问题。例如：

```
$where = index($big, $small);
```

Perl会在$big字符串中寻找$small字符串首次出现的地方，并返回一个整数表示第一个字符的匹配位置，返回的字符位置是从零算起的。如果子字符串是在字符串最开始的位置找到的，那么index会返回0；如果在第二个字符，则返回1；如果index无法找到子串，就会返回-1 [注1]。在这个例子里，$where会得到6：

```
my $stuff = "Howdy world!";
my $where = index($stuff, "wor");
```

另一种理解位置的方法，就是把它当成要走到子字符串之前需要跳过的字符数。因为$where是6，所以我们知道，必须跳过$stuff的前6个字符，才会走到wor。

注1：　之前使用C的程序员肯定会注意到这类似于C语言的index函数。如今的C程序员也可能知道，但此时要想真正理解目前的话题，你最好是那个前C程序员。

index函数每次都会返回首次出现子字符串的位置。不过，你可以再加上可选的第三个参数来指定开始搜索的地方，这样index就不会从字符串的最开头寻找，而是从该参数指定的位置开始寻找子字符串：

```
my $stuff  = "Howdy world!";
my $where1 = index($stuff, "w");               # $where1为2
my $where2 = index($stuff, "w", $where1 + 1);  # $where2为6
my $where3 = index($stuff, "w", $where2 + 1);  # $where3为-1（没找到）
```

（当然，要重复搜索某个子字符串时通常会使用循环。）第三个参数相当于可能的最小返回值，如果子字符串无法在该位置或其后被找到，那么index就会返回-1。

偶尔你会需要搜索子字符串最后出现的位置。这项信息可以用rindex函数来取得，它会从字符串末尾的地方开始找起。在下面的例子中，可以找到最后一个斜线，它在字符串中的位置是4，这和index返回的结果是一样的：

```
my $last_slash = rindex("/etc/passwd", "/");  # 值为4
```

rindex函数也有可选的第三个参数，但这里是用来限定返回值的上限：

```
my $fred = "Yabba dabba doo!";

my $where1 = rindex($fred, "abba");  # $where1 为 7
my $where2 = rindex($fred, "abba", $where1 - 1);  # $where2为1
my $where3 = rindex($fred, "abba", $where2 - 1);  # $where3为-1
```

用substr操作子字符串

substr函数只处理较长字符串中的一小部分内容，大致用法如下：

```
my $part = substr($string, $initial_position, $length);
```

它需要三个参数：一个原始字符串、一个从零起算的起始位置（类似index的返回值）以及子字符串的长度。找到的子字符串会被返回：

```
my $mineral = substr("Fred J. Flintstone", 8, 5);  # 返回"Flint"
my $rock = substr "Fred J. Flintstone", 13, 1000;  # 返回"stone"
```

在上面的例子里你可能已经注意到了，假如所要求的子字符串的长度（此例为1000个字符）超出字符串的结尾，Perl不会抱怨，只不过你会得到比预期长度（1000）短的字符串。如果你想要一直取到字符串结尾，那么不论字符串长短，只要省略第三个参数（子字符串长度）就行了。做法如下：

```
my $pebble = substr "Fred J. Flintstone", 13;  # 返回"stone"
```

一个较大字符串中子字符串的起始位置可以为负值，表示从字符串结尾开始倒数（比如，位置-1就是最后一个字符）[注2]。在下面的例子中，位置-3是从字符串结尾算起的第三个字符，也就是字母i的位置：

```
my $out = substr("some very long string", -3, 2);  # $out 为 "in"
```

如你所愿，index与substr可以紧密合作。在下面的例子里，我们会取出以字符l的位置开头的子字符串：

```
my $long = "some very very long string";
my $right = substr($long, index($long, "l") );
```

接下来就很有意思了：假如原始字符串放在变量里面，我们就可以修改该字符串被选取的部分内容[注3]：

```
my $string = "Hello, world!";
substr($string, 0, 5) = "Goodbye";  # $string 现在的值为 "Goodbye, world!"
```

如你所见，用来取代的（子）字符串的长度并不一定要与被取代的子字符串的长度相同，字符串会自行调整长度。如果这样还不能让你眼前一亮，你还可以用绑定操作符（=~）只对字符串的某部分进行操作。下面的例子只会处理字符串的最后20个字符，将所有的fred替换成barney：

```
substr($string, -20) =~ s/fred/barney/g;
```

substr与index能办到的事多半也能用正则表达式办到。所以，请选择最适合解决问题的方法。但是substr与index通常会快一点，因为它们没有正则表达式引擎的额外负担：它们总是区分大小写，它们不必担心元字符，而且也不会动用任何内存变量。

如果不想给substr函数赋值（因为乍看之下会觉得有点奇怪），你也可以通过传统的[注4]4个参数的方法来使用substr，其中第4个参数是替换子字符串：

注2：这与第三章中介绍的数组索引方式类似。数组应该由0开始（表示第一个元素）往上数，或是从-1（表示最后一个元素）开始往下数。子字符串的位置也是从0开始（表示第一个字符）往上数，或是从 -1（表示最后一个字符）往下数。

注3：从技术上来讲，只要是左值lvalue都行。但此名词的精确定义超出了本书的范围，你可以简单地想成所有能放在赋值号（=）左边的东西。通常该处放的都是变量，但是如同这里所示，也可以放substr操作符。

注4：这里的“传统”指的是“函数调用”的传统风格，而不是指Perl自己的传统风格。因为这个写法是不久之前才引进Perl的。

```
my $previous_value = substr($string, 0, 5, "Goodbye");
```

返回值是替换前的子串。当然，你总是能在空上下文中使用这个函数，不用关心它的返回值。

用sprintf格式化字符串

sprintf函数与printf有相同的参数（可选的文件句柄参数除外），但它返回的是所请求的字符串，而不会直接打印出来。此函数方便之处在于，你可以将格式化后的字符串存放在变量里以便稍后使用。此外，你也可以对结果进行额外的处理，而单靠printf是做不到这些的：

```
my $date_tag = sprintf
    "%4d/%02d/%02d %2d:%02d:%02d",
    $yr, $mo, $da, $h, $m, $s;
```

在上面的例子里，$data_tag会得到类似"2038/01/19 3:00:08"的结果。我们看到，格式字符串（即sprintf函数的第一个参数）在某些格式化数字前加上零，这种用法我们在第五章中介绍printf时并未提及。格式化定义中数字字段的前置零表示必要时会在数字前补零以符合指定的宽度要求。如果没有前置零，那么日期与时间字符串里的数字就不会用零补足宽度，而只留下相应长度的空格，比如"2038/ 1/ 19 3: 0: 8"。

用sprintf格式化金额数字

sprintf的一种常见用法就是格式化小数点后具有特定精度的数字，比如想把2.49997这个金额显示成2.50，而不是2.5，我们可以使用"%.2f"这个字符串轻松完成格式化：

```
my $money = sprintf "%.2f", 2.49997;
```

四舍五入能够使得数字精简并且易读，但是绝大多数情况下应该保留数字的精度，只在输出时做四舍五入运算。

如果手头有个“金额数字（money number）”非常大，大到需要使用逗号来分隔才能有效阅读，那么下面这个子程序能提供很多便利[注5]：

```
sub big_money {
    my $number = sprintf "%.2f", shift @_;
    # 下面的循环中，每次在匹配到的合适位置加一个逗号
```

注5：　是的，我们也知道并非全世界的数字都是三个数字一组，也不是全世界的数字都用逗号分隔，也不是全世界都用美元符号作为货币符号。不过这是一个不错的例子，不必计较细节。

```
    1 while $number =~ s/^(-?\d+)(\d\d\d)/$1,$2/;
    # 在正确的位置补上美元符号
    $number =~ s/^(-?)/$1\$/;
    $number;
}
```

这个子程序用了一些前面没见过的技巧，但不难推测其含义。子程序的第一行是在对第一个（也是唯一的）参数进行格式化，让它在小数点之后刚好有两位数字。也就是说，假如参数是12345678.9，那么$number就会是"12345678.90"。

程序代码的下一行用到了while修饰符。就像之前第十章,其他控制结构中介绍修饰符时提到的，我们可以将它改写成一个传统while循环：

```
while ($number =~ s/^(-?\d+)(\d\d\d)/$1,$2/) {
    1;
}
```

这到底是在做什么？其实这里的意思是，只要这个替换运算返回真（表示成功），就去执行循环主体。但是循环里的程序没有任何作用！对Perl来说，这没关系，这不过是要让我们知道，该语句的主要目的是在执行条件表达式（替换运算），而不是这个无用的循环主体。这么做时习惯上会使用数值1来作为占位符，其实任何数值都可以使用[注6]。下面这行程序代码与上面的循环有相同的效果：

```
'keep looping' while $number =~ s/^(-?\d+)(\d\d\d)/$1,$2/;
```

所以，现在我们知道替换运算是该循环的真正目的。但此处的替换运算又做了什么事？别忘了，这里的$number是个像"12345678.90"这样的字符串。上面的模式会匹配字符串的第一个部分，但它不会越过小数点（你知道为什么吗？）。内存变量$1会是"12345"，而$2会是"678"，所以这个替换运算会让$number变成"12345,678.90"（别忘了，它不会匹配小数点，所以字符串后面的部分并不会改变）。

你知道模式开头的减号有什么用处么？（提示：这个减号只能在字符串中的一个地方出现），在本节结束时，万一你还没找到答案，可以看看我们的解答。

整个替换过程并非到此为止。既然替换运算成功了，这个无事可做的循环就会再来一次。因为这次模式无法匹配到逗号之后的任何东西，所以$number会变成"12,345,678.90"。于是在每次循环执行后，替换运算就会在数字中加一个逗号。

循环的工作还没完呢。因为上一次替换成功了，所以又会重新开始循环。但由于模式必

注6：　也就是说，随便写什么都一样。顺带一提，Perl会对常量表达式进行优化，因此不会花任何运行时间。

须匹配从字符串开头的至少4个数字，所以这一次它无法匹配任何东西，于是就退出循环。

为什么我们不直接使用/g修饰符来进行“全局”查找与替换，以省下1 while循环造成的麻烦与混淆呢？这是因为必须从小数点倒回处理，而不是从字符串开头依次处理，所以不能这么做。像这样将逗号插入一个数字里，只用s///g是没办法做到的[注7]。你想到减号的作用了吗？它让程序也可以处理负号开始的数值字符串。程序代码的下一行也有一样的效果：它会把美元符号放在正确的地方，所以$number会变成"$12,345,678.90"；如果是负数，则会变成"-$12,345,678.90"。请注意，美元符号不一定会是字符串的首字母，不然这行代码会简单许多。最后，结尾的程序代码会返回我们已经格式化得漂漂亮亮的“金额数字”，可以送到年度报表里打印了。

非十进制数字字符串的转换

如果有一个以其他进制表示的数字字符串，我们可以用hex()或oct()函数将字符串转换为对应的数字。有一点非常赞，如果字符串是以特定的十六进制或二进制前缀字符开头的话，oct()函数能聪明地根据该进制进行转换，不过十六进制字符串必须以0x开头，请参考具体例子：

```
hex('DEADBEEF')      # 即十进制的3_735_928_559
hex('OxDEADBEEF')    # 也是十进制的3_735_928_559

oct('0377')          # 十进制的255
oct('377')           # 也是十进制的255
oct('OxDEADBEEF')    # 是十进制的3_735_928_559，因为看到了开头的0x
oct('0b1101')        # 是十进制的13，因为看到了开头的0b
oct("0b$bits")       # 将变量$bits中的值当作二进制数字转换到十进制
```

高级排序

之前在第三章中介绍如何了利用内置的sort操作符，以ASCII码序对列表排序。但如果你希望按数字大小进行排序，或以不区分大小写的方式进行排序呢？你也许还想按照存储在哈希内的信息进行排序。Perl能以任何需要的顺序来为列表排序。接下来一直到本章末，我们会介绍所有有关排序的主题以及实际的例子。

Perl允许你建立自己的“排序规则子程序（*sort-definition subroutine*）”，或简称“排序子程序（*sort subroutine*）”，来实现自定义的排序方式。乍听到“排序子程序”这个术

注7：　至少还得使上几招我们从没教过你的正则表达式技巧，才能完成任务。那么要命的Perl开发小组就知道让书越来越难写，所以我们只好一直说“不行”。

语，如果你上过任何一门计算机科学的课程，脑海中就可能会浮现出冒泡排序、希尔排序和快速排序。然后你会说："拜托，别再谈这些了！"请放心，事情没那么复杂，其实还相当简单。Perl其实知道怎么对列表排序，它只是不知道要用什么样的规则，所以排序子程序只是用来说明具体的规则。

为什么会有这样的需求？仔细想想，排序其实就是比较一堆东西，然后将它们按照特定规则排好队。由于不可能一次比较所有东西，所以最终一定都是两两相比，并根据两者间的顺序进行定位，最终让全体成员就位。Perl已经知道这些步骤了，只是不知道你要如何确定两者的顺序，而这就是需要由你来写的程序。

这也就是说，排序子程序并不需要排序许多元素，只要能比较两个元素就行。只要它能确定两个元素的顺序，Perl就有办法（通过不断咨询排序子程序）返回排好序的数据。

排序子程序的定义和普通的子程序几乎相同。它会被反复调用，每次都会检查要排序列表中的两个元素。

假如子程序要比较两个待排序的参数，你可能会先写出如下代码：

```
sub any_sort_sub {    # 实际上这么写不能正确工作，这里只是为了方便说明问题
    my($a, $b) = @_;  # 声明两个变量并给它们赋值
    # 在这里开始比较$a和$b
    ...
}
```

但请注意，排序子程序会一次次地被调用，往往会运行上百或上千次，这取决于被排序数据的规模。在子程序开始的地方声明变量$a与$b并给它们赋值看起来只会花一丁点时间，但把这些时间乘以排序子程序被调用的次数，就可能非常可观了，这对整体性能会造成不小影响。

我们并不会这么做，事实上这么做是行不通的。其实在子程序开始之前，Perl已经帮我们办好了这些事。实际写出的排序子程序并不会有前面例子的第一行，$a与$b都已经被自动赋值好了。当排序子程序开始运行时，$a与$b会是两个来自原始列表的元素。

子程序会返回一个数字，用来描述两个元素之间的比较（就像C语言中qsort(3)所做的，但它是Perl自己内部的排序实现。假如在结果列表中$a应该在$b之前，排序子程序就会返回-1；如果$b应该在$a之前，它就会返回1。

如果$a与$b无所谓谁先谁后，则该子程序会返回0。为什么会有这样的情况呢？也许你正在进行一个不区分大小写的排序，而被比较的两个字符串是fred和Fred；或者，你正在进行一个按数字大小的排序，而这两个数字恰好相等。

现在我们可以写出下面这样的排序子程序：

```
sub by_number {
    # 排序子程序，使用$a和$b这两个变量进行比较
    if ($a < $b) { -1 } elsif ($a > $b) { 1 } else { 0 }
}
```

要使用这个排序子程序，请把它的名称（去掉与号）写在sort关键字与待排序的列表之间。下面的例子会把按数字大小排序好的结果列表放进@result里：

```
my @result = sort by_number @some_numbers;
```

我们将这个排序子程序命名为by_number，因为这个名称描述了它的排序方式。不过更重要的是，你可以将这一行用来排序的程序代码读成"sort by number（按数字排序）"，就好像在说英语一样。许多排序子程序的名称都是以by_开头的，用以描述它们的排序方式。类似地，我们也可以把排序子程序命名为numerically，不过这样要输入更多字符，而且还比较容易写错。

请注意，在排序子程序中，我们并不需要花力气声明并设定$a与$b，如果我们这么做，该子程序反而无法正常运作。请让Perl帮忙设定$a与$b，我们只要编写它们的比较方式。

事实上，我们可以让它更简单高效。因为常常需要用到这样的三路比较，所以Perl提供了一个方便的简写，飞船操作符（<=>）[注8]。这个操作符会比较两个数字并返回-1、0或1，好让它们依数字排序。所以，我们可以把那个排序子程序写得更好看些，如下所示：

```
sub by_number { $a <=> $b }
```

因为飞碟操作符只能用来比较数字，所以你也许猜到了，会有另一个相应的三路字符串比较操作符：cmp。这两个操作符非常易记，而且一目了然。飞船操作符与数字比较类的操作符>=一脉相承，它之所以由三个字符组成，是因为它会返回三种比较结果，而两个字符的操作符则返回两种结果。同样地，cmp操作符与字符串比较类的操作符ge一脉相承，它也是返回三种比较结果的操作符[注9]。当然，cmp所提供的顺序与sort默认的排序规则相同。所以你不必自己编写下列子程序，因为它和sort默认的排序方法相同[注10]：

注8：　这么称呼是因为《星球大战（Star Wars）》里面的某种钛战机就是这样。起码我们觉得很像。

注9：　这并非偶然。Larry这么做是为了让Perl更好学，也更好记忆。他骨子里就是个语言学家，所以他深知学习语言的痛苦。

注10：　除非你也在写一本Perl入门书并且需要举例说明，否则永远不会需要自己写。

```
sub by_code_point { $a cmp $b }

my @strings = sort by_code_point @any_strings;
```

不过，cmp可以用来建立更复杂的排序顺序，例如不区分大小写的排序：

```
sub case_insensitive { "\L$a" cmp "\L$b" }
```

这个例子中，我们比较了$a里的字符串（强制转换成小写）和$b里的字符串（强制转换成小写），以构造不区分大小写的排序顺序。

不过，你要知道Unicode字符串里面的字符有时可以用不同的形式表示，为了按照实际的字符意义排序，我们需要先把某种表示形式转化为统一的形式后再进行比较排序。有关这方面的知识，我们放在附录 C介绍。所以一般来说，要对Unicode字符串排序，都会写成下面这种样子：

```
use Unicode::Normalize;

sub equivalents { NFKD($a) cmp NFKD($b) }
```

另外需要注意的是，在比较过程中我们并没有修改被比较元素的值[注11]，我们只是用它们的值比大小而已。这一点很重要：出于性能的考虑，$a和$b并非数据项拷贝，实际上它们只是原始列表元素的临时别名。所以，如果中途改变它们的值，就会弄乱原始数据。千万别这么做，Perl不支持也不建议这种行为。

如果排序子程序像我们的例子一样简单（大部分情况下都很简单），你就可以让程序代码更为简单，只要把整个排序子程序内嵌到排序子程序名的位置就行了。做法如下：

```
my @numbers = sort { $a <=> $b } @some_numbers;
```

事实上，在如今的Perl开发界，几乎不会有人写额外的排序子程序，你常常会看到这种内嵌的排序子程序。

假设要以递减的顺序进行排序，reverse函数可以帮助我们轻松达成：

```
my @descending = reverse sort { $a <=> $b } @some_numbers;
```

这里有个小窍门。比较操作符（<=>与cmp）是短视的，也就是说，它们并不知道哪个操作数是$a，哪个操作数是$b，只知道哪一个在左边，哪一个在右边。所以，如果我们把$a与$b对调，比较操作符每次就会得到相反的结果。换句话说，下面这么做也能得到反向排序的结果：

注11： 除非调用的比较子程序内部对比较的内容作出修改，不过那种情况十分罕见。

```
my @descending = sort { $b <=> $a } @some_numbers;
```

稍加练习之后，你就可以一眼看出这一行在做什么。它是递减比较（因为$b在$a之前，也就是递减的顺序），而且是数字比较（因为它使用飞船操作符，而不是cmp）。所以，它代表以反向顺序进行的数字排序。在较新的Perl版本中，这两种方法并没有太明显的差别，因为reverse已经被当成是sort的一个修饰符了，在处理时会做特殊优化，以避免多余的逆序操作。

按哈希值排序

当你快乐地对列表排序一段时间后，就会碰到按哈希值进行排序的问题。例如，本书里曾出现过的三个主角昨晚跑去打保龄球，他们的积分存储在下面的哈希里。我们希望能用适当的顺序将名字打印出来，也就是积分最高的人在最上面。所以我们需要按积分来对此哈希排序：

```
my %score = ("barney" => 195, "fred" => 205, "dino" => 30);
my @winners = sort by_score keys %score;
```

当然，我们实际上无法按积分来对哈希排序，这只是口头上的说法而已。哈希本身是没法排序的！虽然我们之前用sort对哈希键排序，但其实是对哈希键的名称进行排序（以ASCII码序）。现在我们仍要对哈希键排序，但现在的顺序是以它们对应的哈希值为基准。在这个例子里，我们需要的结果是：三个主角名的列表，按照他们的保龄球积分进行排序。

这个排序子程序相当容易实现。我们要的是积分的数值比较，而不是名字。换句话说，我们不该比较$a与$b这两个球员名字，而是比较$score{$a}与$score{$b}这两个积分。如果你也想到这点了，那么答案就呼之欲出了。如下所示：

```
sub by_score { $score{$b} <=> $score{$a} }
```

让我们来仔细分析其运作原理。假如它第一次被调用时，Perl把$a设定为barney，而$b则是fred。所以这次的比较是$score{"fred"}<=>$score{"barney"}，（参考上面的哈希）也就是205<=>195。别忘了飞船操作符是短视的，所以当它看到205在左边，而195在右边时，它就机械地说："不，这不是正确的数值顺序，$b应该在$a之前。"所以它会告诉Perl，fred应该在barney之前。

接下来第二次调用时，$a仍然是barney，而$b则变成了dino。短视的飞船操作符看到30<=>195，就会认为顺序是对的。$a应该在$b之前，也就是barney应该在dino之前。最终Perl得到了排序的结果：fred是冠军，barney是亚军，最后是dino。

为什么这里的比较将$score{$b}放在$score{$a}之前，而不是反过来呢？因为我们想要

将积分按降序排列，由分数最高者依序往下排列。所以你只要稍加练习，就可以一眼看出：$score{$b}<=>$score{$a}表示按积分降序排列。

按多个键排序

我们忘记说了，昨晚除了这三位之外，还有第四位玩家，所以哈希其实应该是下面这样：

```
my %score = (
    "barney" => 195, "fred" => 205,
    "dino" => 30, "bamm-bamm" => 195,
);
```

现在，如你所见，bamm-bamm和barney积分相同。所以在排序完成之后，哪一个应该排在前面呢？我们看不出来，因为比较操作符在比较这两个数字时，发现两边都看到相同的分数，就会返回零。

也许这无关紧要，不过通常我们更喜欢明确定义的排序。如果有多个玩家同分的话，他们当然都应该排在一起，但这些名字应该按ASCII码序排列。这样的排序子程序该怎么写呢？其实也很简单：

```
my @winners = sort by_score_and_name keys %score;

sub by_score_and_name {
    $score{$b} <=> $score{$a}  # 根据分数降序排列
        or
    $a cmp $b                  # 分数相同的再按名字的ASCII码序排列
    }
```

这里是怎么运作的？嗯，如果飞船操作符看到两个不同的数字，那这就是我们想要的比较运算。它会返回-1或1，而这两个都为真，所以在这里低优先级的短路操作符or会将表达式的其余部分跳过，并返回我们所要的比较结果（别忘了，短路操作符or会返回最后执行的表达式的结果）。但是，如果飞船操作符看到两个相同的分数，它会返回0。因为这个值为假，所以让后面的cmp操作符获得执行机会，于是就返回将哈希键作为字符串比较的顺序值。也就是说，如果同分的话，就会按字符串顺序的比较进行最终裁决。

我们知道，使用by_score_and_name这个排序子程序时，它绝不会返回0（知道为什么吗？答案在脚注里）[注12]。所以，我们知道排列的顺序都是明确定义的。也就是说，如

注12： 让其返回0的唯一情况是当两个字符串完全相同时，而现在处理的是哈希的键，我们早就知道它们绝不会相同。如果你在调用sort时传入了一个列表，内有重复（完全相同）的字符串，那么比较这些字符串时就会返回0，但此处传入的是哈希的键，所以不存在重复的情况。

果今天的数据和明天的数据都一样，那么今天和明天的答案也会一样。

当然，没有什么理由限制排序子程序只能做两级排序。下面列出的程序代码能对读者的ID编号列表进行五级排序[注13]。这个例子里的排序是根据每个读者的未缴罚金（用&fines子程序计算，在此未列出）、目前他们借阅的本数（取自%items）、他们的姓名（先按姓排，后按名排，两者都取自哈希），最后是顾客的ID编号，以防前面的信息都相同：

```
@patron_IDs = sort {
  &fines($b) <=> &fines($a) or
  $items{$b} <=> $items{$a} or
  $family_name{$a} cmp $family_name{$b} or
  $personal_name{$a} cmp $family_name{$b} or
  $a <=> $b
} @patron_IDs;
```

习题

以下习题答案参见第332页上的“第十四章习题解答”一节：

1. [10]编写程序，读入一连串数字并将它们按数值大小排序，将结果以右对齐的格式输出。请用下列数据进行测试：

   ```
   17 1000 04 1.50 3.14159 -10 1.5 4 2001 90210 666
   ```

2. [15]编写程序，以不区分大小写的字符顺序把下列的哈希数据按姓氏排序后输出。当姓一样时，再按名排序（还是一样，不区分大小写）。也就是说，输出结果中的第一个名字应该是Fred的，最后一个应该是Betty的。所有姓相同的人应该要排在一起。千万别更改原始数据。这些名字应该以它们原本的大小写形式显示：

   ```
   my %last_name = qw{
     fred flintstone Wilma Flintstone Barney Rubble
     betty rubble Bamm-Bamm Rubble PEBBLES FLINTSTONE
   };
   ```

3. [15]编写程序，在输入字符串中找出指定子字符串出现的位置并将其输出。例如：输入字符串为"Thisisatest."，而子字符串是"is"，则程序应该会汇报位置2和5；如果子字符串是"a"，程序应该会汇报8；如果子字符串是"t"，程序将汇报什么？

注13：　虽然在史前时代很不寻常，但在现代社会里，确实有可能需要进行五级以上的排序。

第十五章

智能匹配与given-when结构

要是计算机自己就能弄明白我们想要做的事并做完它，那该有多好啊！Perl已经尽力去这么做了，在你想要数字的时候给你数字，想要字符串的时候给你字符串，想要单个值的时候给你单个值，想要列表的时候给你列表。加上现在Perl 5.10提供的智能匹配操作符和given-when控制结构，就更完美了。

智能匹配是从Perl 5.10.0开始出现的，当时还存在一些问题。不过到了Perl 5.10.1，绝大多数的bug都得到了修正，所以也不算什么大问题。（这意味着也许你需要为本章更更新一下你的Perl版本。）所以，如果你仍然在用Perl 5.10.0，就不要考虑使用智能匹配了，否则随后可能会招致意外。因此，本章我们会改用更为准确的基础版本号，提醒你使用较新的Perl版本：

```
use 5.010001;  # 至少是 5.10.1 版
```

智能匹配操作符

智能匹配操作符~~会根据两边的操作数的数据类型自动判断该用何种方式进行比较或匹配。如果两边的操作数看起来都像数字，就按数值来比较大小；如果看起来像字符串，就按字符串方式比较；如果某一端操作数是正则表达式，那就当作模式匹配来执行。它还能完成许多复杂任务，如果换成功能相同的传统写法，多半会是一大堆冗繁代码。所以有了它，我们可以省下不少力气。

这个~~看起来和第八章介绍过的绑定操作符=~非常相近。不过~~更能干些。有时候，它甚至就能取代绑定操作符。之前，你已经学会使用绑定操作符关联$name和正则表达式来匹配模式：

```
print "I found Fred in the name!\n" if $name =~ /Fred/;
```

现在，用智能匹配操作符代替绑定操作符也能完成同样的任务：

```
use 5.010001;

say "I found Fred in the name!" if $name ~~ /Fred/;
```

智能匹配操作符看到左侧有个标量，右侧有个正则表达式，于是它自己推断出应该执行模式匹配操作。不算惊天动地，但已经有了长足进步。

在处理更复杂的情况时，智能匹配操作符才会大显身手。比方说，你想在哈希%names中查找任何匹配Fred的键，如果找到就打印一条消息出来。你无法用exists判定，因为它需要给定确切的键。当然，你可以用foreach遍历每个键，尝试用正则表达式匹配，跳过那些不匹配的，直到发现要找的键，保存到变量$flag中，然后用last跳出循环：

```
my $flag = 0;
foreach my $key ( keys %names ) {
    next unless $key =~ /Fred/;
    $flag = $key;
    last;
}

print "I found a key matching 'Fred'. It was $flag\n" if $flag;
```

唷！这么麻烦，连解释起来都很费力。不过这么写也有好处，就是兼容各种Perl 5版本。可有了智能匹配操作符，只要把哈希写在左侧，把正则表达式写在右侧，就搞定了：

```
use 5.010001;

say "I found a key matching 'Fred'" if %names ~~ /Fred/;
```

之所以智能匹配操作符知道该怎么做，是因为它看到了一个哈希和一个正则表达式。遇到这两种操作数时，智能匹配操作符就知道该遍历%names的所有键，用给定的正则表达式逐个测试。如果找到匹配的那个键，它就知道该停下来了并立即返回真。这里的匹配和对标量的匹配不太一样。它很聪明，能因地制宜，知道怎么做才是正确的。并且它集数种不同的操作于一身，仅仅一个操作符就能解决各式各样的问题。

那么，你该如何判断它是怎么工作的呢？在*perlsyn*文档中有一张表，列出了两边出现不同类型的操作数时会采取何种匹配行为。我们现在这个例子中，左边还是右边出现正则表达式或哈希都无关紧要，因为那张智能匹配工作表告诉我们，无论这两者的顺序先后，其匹配行为都是一样的。所以，你也可以倒过来写：

```
use 5.010001;

say "I found a key matching 'Fred'" if /Fred/ ~~ %names;
```

如果想要比较两个数组（为简单起见，暂时只考虑相同长度数组之间的比较），可以按数组索引依次遍历，取出相同位置的两个元素来比较。如果比较下来两者相等，则令计数器$equal自增1。循环结束后，如果$equal的数字和数组@names1的长度一致，就说明这两个数组是完全相同的：

```
my $equal = 0;
foreach my $index ( 0 .. $#names1 ) {
    last unless $names1[$index] eq $names2[$index];
    $equal++;
}

print "The arrays have the same elements!\n"
    if $equal == @names1;
```

还是要说，这么做太麻烦了。就不能有更轻松一点的实现方法吗？等等！智能匹配操作符如何？直接把两个数组放在~~的两端。单单下面这点代码就做了和上面一样的事情，而且看起来几乎没什么要写的代码：

```
use 5.010001;

say "The arrays have the same elements!"
    if @names1 ~~ @names2;
```

好吧，再来看一个例子。假设在调用了某个函数之后，你想要检查它的返回值是否在某个可能或预期值集合中。回想第四章中介绍的max()子程序，你已经知道max()将会返回传给它的值中最大的那个。你可以将max的返回值和传给它的参数列表作比较，还是用之前的那种老式笨拙的写法：

```
my @nums   = qw( 1 2 3 27 42 );
my $result = max( @nums );

my $flag = 0;
foreach my $num ( @nums ) {
    next unless $result == $num;
    $flag = 1;
    last;
}

print "The result is one of the input values\n" if $flag;
```

你已经知道我们会怎么说：这么做太麻烦啦！把中间那些代码都扔掉吧，改用~~好了。这就容易多了：

```
use 5.010001;

my @nums   = qw( 1 2 3 27 42 );
my $result = max( @nums );
```

```
say "The result [$result] is one of the input values (@nums)"
    if @nums ~~ $result;
```

智能匹配对两边操作数的顺序一般没有要求，倒过来写也行。对这里的例子中两边的数据类型来说，智能匹配操作符不关心谁在哪一边：

```
use 5.010001;

my @nums   = qw( 1 2 3 27 42 );
my $result = max( @nums );

say "The result [$result] is one of the input values (@nums)"
    if $result ~~ @nums;
```

智能匹配操作的优先级

现在你已经看到了，智能匹配操作符是如此简洁灵巧。如果还需要进一步了解各种情况下它会如何工作，请仔细查看*perlsyn*文档中"Smart matching in detail"部分的表格。下面的表15-1列出了一部分。

表15-1：智能匹配操作符对不同操作数的处理

范例	匹配类型
%a ~~ %b	哈希的键是否一致
%a ~~ @b或@a ~~ %b	%a中的至少一个键在列表@b中
%a ~~ /Fred/或/Fred/ ~~ %b	至少有一个键匹配给定的模式
'Fred' ~~ %a	是否存在$a{Fred}
@a ~~ @b	数组是否相同
@a ~~ /Fred/	@a中至少有一个元素匹配模式
$name ~~ undef $name	$name没有定义
$name ~~ /Fred/	模式匹配
123 ~~ '123.0'	数值和"numish"类型的字符串是否相等
'Fred' ~~ 'Fred'	字符串是否相同
123 ~~ 456	数值是否相等

当使用智能匹配操作符时，Perl会按此表自上而下查看适用的操作数配对。先找到哪一种搭配就选择哪一种操作。操作数的顺序偶尔会有区别。假如你有一个数组和一个哈希要进行智能匹配：

```
use 5.010001;
```

```
if ( @array ~~ %hash ) { ... }
```

Perl先去找对应于一个数组和一个哈希的匹配类型，结果是要求@array中至少有一个元素是哈希%hash的键。这很容易，因为对于这两种操作数类型，只有一种匹配类型。

智能匹配操作符并非始终符合交换律。说起交换律，你可能会想起高中代数中的术语，它的意思是操作数的顺序无关，颠倒后的计算结果仍然一样。智能匹配有时不满足交换律，当看到左侧是数字时，就会按照数值方式进行比较，看到左侧是字符串时，就会按照字符串方式进行比较。所以同样的两个数字和字符串，按不同顺序做智能匹配，得到的结果是不同的，关键还是看它第一个拿到的操作数的类型，也就是左侧数据：

```
use 5.010001;

say "match number ~~ string" if 4 ~~ '4abc';
say "match string ~~ number" if '4abc' ~~ 4;
```

执行后只有一个智能匹配有输出：

```
match string ~~ number
```

虽然第一个智能匹配的左侧是数字类型，但它实际上却是字符串比较。在优先级表中，唯一一条左侧是数字类型操作数的，右侧期望的是看起来像数字的字符串（numish）类型操作数。而这里的4abc对Perl来说并不足以判为这种numish类型。所以不符合左侧数字右侧numish类型的条目，于是智能匹配操作符顺势往下，一直到最后一条“Any”对“Any”的条目，而该条目是按照字符串方式进行比较，显然两者并不匹配，没有输出。

第二个智能匹配是按照数字方式比较。它的左侧是“Any”类型而右侧是“Num”类型，这个条目的优先级要比第一个智能匹配所用的条目高上好几级。它按照数值方式比较，左侧字符串自动转换为数字后和右侧相等，所以两者匹配，打印输出。

那么，如果有两个标量变量的话，该如何匹配呢？

```
use 5.010001;

if ( $fred ~~ $barney ) { ... }
```

就目前而言，Perl还无法判断该用何种匹配方式，它需要知道标量变量$fred和$barney里面的数据到底是什么类型。Perl无法预先判断，要一直等到取得具体数据为止。这里的智能匹配究竟是按数值方式比较还是按字符串方式比较？我们现在还无从知晓。

given语句

given-when控制结构能够根据given后面的参数执行某个条件对应的语句块。这是Perl用来应对C语言的switch语句的等效物，只不过更具Perl色彩，所以更时尚，名字也更新潮些。

下面这段代码从命令行取第一个参数，$ARGV[0]，然后依次走一遍when条件测试，看看是否可以找到Fred。每个when语句块都对应于不同的处理方式，我们从最宽松的条件开始，测试并报告Fred出现的情况：

```
use 5.010001;

given ( $ARGV[0] ) {
    when ( 'Fred'   )  { say 'Name is Fred' }
    when ( /fred/i )  { say 'Name has fred in it' }
    when ( /\AFred/ ) { say 'Name starts with Fred' }
    default           { say "I don't see a Fred" }
}
```

given会将参数化名为$_ [注1]，每个when条件都尝试用智能匹配对$_作测试。当然为了概念清晰，你可以用显式智能匹配的方式改写上面的例子：

```
use 5.010001;

given( $ARGV[0] ) {
    when ( $_ ~~ 'Fred'  )  { say 'Name is Fred' }
    when ( $_ ~~ /\AFred/ ) { say 'Name starts with Fred' }
    when ( $_ ~~ /fred/i )  { say 'Name has fred in it' }
    default                 { say "I don't see a Fred" }
}
```

如果$_不能满足任何一个when条件，Perl就会执行default语句块。下面是若干次运行后的输出结果：

```
$ perl5.10.1 switch.pl Fred
Name is Fred
$ perl5.10.1 switch.pl Frederick
Name starts with Fred
$ perl5.10.1 switch.pl Barney
I don't see a Fred
$ perl5.10.1 switch.pl Alfred
Name has fred in it
```

你可能会说：“不稀奇，我可以用if-elsif-else来写这个例子。”接下来的例子正是用

注1：　按Perl的术语来说，given是一个主题符（*topicalizer*），因为它能使后面的参数成为主题（topic），而它是Perl 6当中给$_起的时尚新名字。

这种方式写的，而且还用my来声明当前语句块的私有词法变量$_，这是Perl 5.10里面my的新用法[注2]：

```
use 5.010001;

{
my $_ = $ARGV[0]; # 从 Perl 5.10 开始，$_ 可以声明为词法变量了！

   if ( $_ ~~ 'Fred'  ) { say 'Name is Fred' }
elsif ( $_ ~~ /\AFred/ ) { say 'Name starts with Fred' }
elsif ( $_ ~~ /fred/i )  { say 'Name has fred in it' }
else                     { say "I don't see a Fred" }
}
```

如果given能做的和if-elsif-else完全一样的话，它就没必要存在了。和if-elsif-else不同，given-when可以在满足某个条件的基础上继续测试其他条件，而if-elsif-else一旦满足了某个条件，就只能执行对应的那个语句块。

在继续深入之前，还是来看看显式写法的例子，搞清楚究竟发生了些什么。除非明确指定，否则在when语句块后面都好像有一句break，它告诉Perl现在就跳出given-when结构，继而执行后面的程序。之前的例子真好像带有break似的，尽管不需要你自己输入：

```
use 5.010001;

given ( $ARGV[0] ) {
    when ( $_ ~~ 'Fred'  )  { say 'Name is Fred'; break }
    when ( $_ ~~ /fred/i )  { say 'Name has fred in it'; break }
    when ( $_ ~~ /\AFred/ ) { say 'Name starts with Fred'; break }
    default                 { say "I don't see a Fred"; break }
}
```

这种写法用在以上的例子中并不太合适。因为我们是按从宽泛到特殊的顺序来测试的。如果传来的参数匹配/fred/i，Perl就不会测试其余的when条件了。同样的道理，我们不能一开始就测试是否包含Fred，因为这样就总没有机会做后面的测试，执行到第一个when语句块后Perl就会跳出该控制结构。

如果在when语句块的末尾使用continue，Perl就会尝试执行后续的when语句了。这就是if-elsif-else力不能及的地方。当另一个when的条件满足时，Perl会执行对应语句块（同样，默认退出整个控制结构，除非你写清楚）。在每个when语句块的末尾写上continue，就意味着所有的条件测试都会执行：

注2： Perl的特殊变量都是全局变量，这一点我们之前在第四章提过。但$_变量太受欢迎了，所以Perl 5.10开始允许我们将它变为词法变量。这样一来，在你指定的语句块内部，默认会用到$_的内置函数或操作符就不会受外部数据的影响了。

```
use 5.010001;

given ( $ARGV[0] ) {
    when ( $_ ~~ 'Fred'   )  { say 'Name is Fred'; continue } # 糟糕!
    when ( $_ ~~ /fred/i )  { say 'Name has fred in it'; continue }
    when ( $_ ~~ /\AFred/ ) { say 'Name starts with Fred'; continue }
    default                 { say "I don't see a Fred" }
}
```

这里还有一个小问题。我们注意到default块总是会运行：

```
$ perl5.10.1 switch.pl Alfred
Name has fred in it
I don't see a Fred
```

default块就相当于一个测试条件永远为真的when语句。如果在default之前的when语句使用了continue，Perl就会继续执行default语句。所以本质上，default就是另一个when：

```
use 5.010001;

given ( $ARGV[0] ) {
    when ( $_ ~~ 'Fred'   )  { say 'Name is Fred'; continue }
    when ( $_ ~~ /\AFred/ ) { say 'Name starts with Fred'; continue }
    when ( $_ ~~ /fred/i )  { say 'Name has fred in it'; continue } # 糟糕!
    when ( 1 == 1        )  { say "I don't see a Fred" } # 相当于default语句
}
```

要解决这个问题，只要拿掉最后一个的continue，这样最后一个when就会停止进程：

```
use 5.010001;

given ( $ARGV[0] ) {
    when ( $_ ~~ 'Fred'   )  { say 'Name is Fred'; continue }
    when ( $_ ~~ /\AFred/ ) { say 'Name starts with Fred'; continue }
    when ( $_ ~~ /fred/i )  { say 'Name has fred in it' }  # 现在好了!
    when ( 1 == 1        )  { say "I don't see a Fred" }
}
```

现在我们已经都交代清楚了，下面再用常规写法改写，以后你也可以将这种形式用到实际的程序中：

```
use 5.010001;

given ( $ARGV[0] ) {
    when ( 'Fred'   )  { say 'Name is Fred'; continue }
    when ( /\AFred/ ) { say 'Name starts with Fred'; continue }
    when ( /fred/i )  { say 'Name has fred in it'; }
    default           { say "I don't see a Fred" }
}
```

笨拙匹配

除了在given-when中使用智能匹配外，还可以像以前那样使用所谓的“笨拙（dumb）”匹配。当然，这并不是真的说它笨拙，我们指的是你之前已经掌握的正则表达式匹配方式。Perl只要看到明确书写的比较操作符（不管哪种类型）或是绑定操作符，它就会按这些操作符的要求去做：

```
use 5.010001;

given ( $ARGV[0] ) {
    when ( $_ eq 'Fred'   )  { say 'Name is Fred'; continue }
    when ( $_ =~ /\AFred/ ) { say 'Name starts with Fred'; continue }
    when ( $_ =~ /fred/i )  { say 'Name has fred in it'; }
    default                 { say "I don't see a Fred" }
}
```

你甚至可以混用笨拙匹配和智能匹配，每个when表达式都会各自处理不同类型的匹配方式：

```
use 5.010001;

given ( $ARGV[0] ) {
    when ( 'Fred'   )       { # 智能匹配
        say 'Name is Fred'; continue }
    when ( $_ =~ /\AFred/ ) { # 笨拙匹配
        say 'Name starts with Fred'; continue }
    when ( /fred/i )        { # 智能匹配
        say 'Name has fred in it'; }
    default                 { say "I don't see a Fred" }
}
```

注意笨拙和智能这两种匹配方式并非泾渭分明，因为正则表达式的匹配操作符默认也是用$_来作测试的。

智能匹配操作符用于判断事物是否相同（或者差不多是相同），所以在需要比较大小时，就不能用智能匹配了。此时还是直接用传统的比较操作符好了：

```
use 5.010001;

given ( $ARGV[0] ) {
    when ( ! /\A-?\d+\.\d+\z/ ) { # 笨拙匹配
        say 'Not a number!' }
    when ( $_ > 10 )            { # 笨拙匹配
        say 'Number is greater than 10'  }
    when ( $_ < 10 )            { # 笨拙匹配
        say 'Number is less than 10' }
    default                     { say 'Number is 10' }
}
```

某些特定情况下，Perl会自动使用笨拙匹配方式。若在when里调用某个子程序[注3]，Perl会根据返回值的真假作判断：

```
use 5.010001;

given ( $ARGV[0] ) {
    when ( name_has_fred( $_ ) ) { # 笨拙匹配
        say 'Name has fred in it'; continue }
}
```

同样的规则还适用于Perl的内置函数，比如defined、exists和eof，因为这些函数本来就是要返回真假值的。

否定的表达式，包括否定的正则表达式，都不会使用智能匹配方式。下面的例子和你从上一章看到的类似：

```
use 5.010001;

given( $ARGV[0] ) {
    when( ! $boolean ) { # 笨拙匹配
        say 'Name has fred in it' }
    when( ! /fred/i  ) { # 笨拙匹配
        say 'Does not match Fred' }
    }
```

多个条目的when匹配

有时候需要遍历许多条目，可given只能一次接受一个参数。当然你可以将given放到foreach里面循环测试。比如，要遍历@names，可以依次将当前元素赋值到$name，然后再用given：

```
use 5.010001;

foreach my $name ( @names ) {
    given( $name ) {
    ...
    }
}
```

猜我想说啥？没错，这么写太啰嗦了。难道你还没厌倦这种额外多出来的工作么？这回，直接用@names中当前元素的别名好了，让given直接用这个别名的别名作为参数。Perl就该这样聪明！别担心，它实际上就是这样聪明。

注3：　Perl对方法调用也不会使用智能匹配，不过面向对象编程的内容在《Intermediate Perl》一书中另有介绍。

要遍历多个元素，可以直接省略given，让foreach将当前正在遍历的元素放入它自己的$_里。因为接下来我们要用智能匹配，所以当前元素就只能是放在$_里面：

```
use 5.010001;

foreach ( @names ) { # 不要使用具名控制变量!
    when ( /fred/i )  { say 'Name has fred in it'; continue }
    when ( /\AFred/ ) { say 'Name starts with Fred'; continue }
    when ( 'Fred'   )  { say 'Name is Fred'; }
    default           { say "I don't see a Fred" }
}
```

如果要遍历多个元素的名称，我们总希望可以看到当前遍历到的元素的名称。可以在foreach语句块中写上其他语句，比如say：

```
use 5.010001;

foreach ( @names ) { # 不要使用具名控制变量!
    say "\nProcessing $_";

    when ( /fred/i )  { say 'Name has fred in it'; continue }
    when ( /\AFred/ ) { say 'Name starts with Fred'; continue }
    when ( 'Fred'   )  { say 'Name is Fred'; }
    default           { say "I don't see a Fred" }
}
```

你甚至还可以在若干when语句之间写上其他语句，比方说，在default之前加上一条输出调试信息的语句（在given结构里也可以这么用哦）：

```
use 5.010001;

foreach ( @names ) { # 不要使用具名控制变量!
    say "\nProcessing $_";

    when ( /fred/i )  { say 'Name has fred in it'; continue }
    when ( /\AFred/ ) { say 'Name starts with Fred'; continue }
    when ( 'Fred'   )  { say 'Name is Fred'; }
    say "Moving on to default...";
    default           { say "I don't see a Fred" }
}
```

习题

以下习题答案参见第335页上“第十五章习题解答”一节：

1. [15]重写第十章中的第一道习题，也就是猜数字的那题，使用given结构来实现。想想看该如何处理非数值的输入？这里不需要用智能匹配。
2. [15]用given-when结构写一个程序，根据输入的数字，如果它能被3整除，就打

印“Fizz”；如果它能被5整除，就打印“Bin”；如果它能被7整除，就打印“Sausage”。比如输入数字15，程序该打印“Fizz”和“Bin”，因为15可以同时被3和5整除。思考一下，可以让程序输出“Fizz Bin Sausage”的最小数字该是多少？

3. [15]用for-when写个程序，要求从命令行遍历某个目录下的文件列表，并报告每个文件是否可读、可写或可执行。不需要使用智能匹配。

4. [20]使用given和智能匹配写个程序，从命令行得到一个数字，打印出这个数字除了1和它本身以外的因数。比如输入99，你写的程序该报告3、9、11和33这4个数。如果输入的数字就是一个质数，程序要报告说明这是质数。如果输入的不是数字，也应该报告说明输入有误，不会继续计算。虽然可以用if结构和笨拙匹配来实现，不过作为练习，这里请只用智能匹配来实现。

   作为开始，下面给出一段返回质数的子程序供你使用。它会尝试从2到输入数字$number的一半范围内的数字，看哪些可以被整除：

```
sub divisors {
    my $number = shift;

    my @divisors = ();
    foreach my $divisor ( 2 .. ( $number/2 ) ) {
        push @divisors, $divisor unless $number % $divisor;
    }

    return @divisors;
}
```

5. [20]修改上题程序，报告输入数字的奇偶情况，是否是质数（除1和本身外没有其他可整除的数），是否可以被某个你最喜欢的数字整除。还是只能用智能匹配实现。

第十六章

进程管理

身为程序员最棒的一面，就是能运行别人的程序，不必自己动手去写。现在，我们来学习一下如何从Perl直接运行其他程序并由此学习如何管理这些孩子 [注1]。

在Perl里有句话："办法不止一种"，这里也是如此的。这些办法可能有许多重叠或差异之处并且有各自的特色。如果你不喜欢头一种方法，大可往下读个一两页，找到比较合胃口的方式。

Perl的可移植性非常高。本书的其他章节大都不需要用脚注来说明某个程序在Unix上是这样，在Windows上是那样，而在VMS又是另外一种情况。但当你想在自己的计算机上运行别人的程序时，请注意：在Macintosh上能找到的程序与老式的Cray（曾经是"超级"计算机）上的多半大不相同。本章的例子将以Unix环境为主，如果你使用的不是Unix系统，难免会碰到一些差异。

system函数

在Perl中，启动子进程最简单的方法是用system函数。比如要从Perl调用Unix的*date*命令，需要告诉system要运行的外部程序的名字：

```
system 'date';
```

你所运行的Perl程序称为父（*parent*）进程，当它运行时，system命令根据当前的父进程创建一份拷贝，这份拷贝称为子（*child*）进程。子进程会立即切换到要运行的外部命令上，比如这里的*date*，它继承了原来进程中Perl的标准输入、标准输出以及标准错误。也

注1：　实际上是新生的子进程。

就是说，由外部命令*date*输出的日期与时间字符串会立即传送到当前Perl程序的STDOUT句柄所指向的地方。

通常提供给system函数的参数就是那些一般在shell中键入的命令。所以当你想用*ls-l$HOME*之类比较复杂的命令时，只要把它全部放进参数里就行了：

```
system 'ls -l $HOME';
```

请注意，因为这里的$HOME是shell的环境变量，所以用的不是双引号，而是单引号。否则，因为美元符号同时也是通知Perl的标量符号，所以放在双引号内会进行变量内插，而内插之后shell看到的就不是这个环境变量名了。除此之外，我们还可以这么写：

```
system "ls -l \$HOME";
```

但这样写很快就会乱成一团。

目前*date*命令只是输出结果而已。现在假设它成了比较健谈的命令，会先问“你想知道哪个时区的时间？”[注2]这个询问信息会首先出现在标准输出上，接着程序从标准输入（继承自Perl的STDIN）等待回应。你会看到这个问题，然后键入答案（比如“Zimbabwe time”），然后*date*才能完成它的任务。

子进程正在运行时，Perl会很有耐心地等它结束。如果*date*命令耗时37秒，Perl就会暂停37秒。然而，你可以利用shell提供的功能来启动后台进程[注3]：

```
system "long_running_command with parameters &";
```

现在会启动shell，而它会注意到命令行结尾的与号。于是shell让long_running_command成为后台进程并立即退出。接着，Perl会注意到shell已经返回了，现在可以做别的事情了。在这个例子中，long_running_command其实是Perl的孙（*grandchild*）进程，只是Perl无法直接控制（或访问）它，也不知道它的存在。

当命令“很简单”的时候，不会用到shell。在之前运行*date*与*ls*命令时，Perl会寻找待执行的命令，必要时会使用继承而来的PATH变量[注4]。找到之后，就直接启动它。但如果

注2：　不过据我所知，目前还没有人设计出这种工作方式的*date*命令。

注3：　现在你知道我们所说的“因使用的系统而异”的意思了吗？Unix shell（*/bin/sh*）允许你在运行这类命令时使用与号来启动一个后台进程。当然如果你在非Unix系统上工作，往往不能这样启动后台进程，那样自然也无法这么做。

注4：　PATH是一系列存放可执行程序的目录清单，在非Unix系统上也有这样的概念。你随时可以通过修改Perl的%ENV中的$ENV{'PATH'}的值来达到改变环境变量的目的。初始时，这是从父进程（通常是 shell）继承的环境变量。改变这个值会影响新的子进程，但不能影响到之前的父进程。

字符串里出现奇怪的字符，例如shell元字符：美元符号、分号、竖线等，那么会调用标准Bourne Shell（*/bin/sh*）[注5]来处理。这样一来，shell是子进程，要执行的命令则是孙进程（或是更下一代）。

比如，你可以把下面简短的shell脚本放进参数里[注6]：

```
system 'for i in *; do echo == $i ==; cat $i; done';
```

这里我们再次使用了单引号，因为其中的美元符号是给shell用的，而不应该由Perl解释。双引号会让Perl内插$i的值，而不是用作shell的变量[注7]。附带一提，这个简短的shell脚本会将当前目录中每个文件的文件名及其内容显示出来。如果你不相信我们的话，不妨自己试试。

避免使用Shell

system操作符也可以用一个以上的参数来调用[注8]，如此一来，不管你给的文本有多复杂，都不会用到shell。做法如下[注9]：

```
my $tarfile = 'something*wicked.tar';
my @dirs = qw(fred|flintstone <barney&rubble> betty );
system 'tar', 'cvf', $tarfile, @dirs;
```

在这个例子里，第一个参数'tar'是命令名称，在shell里可以通过PATH环境变量辅助定位它。接下来，后面的参数会被逐项传递给前面的命令。即使参数里出现对shell有意义的字符，例如$tarfile存储的文件名中的星号或者@dirs存储的路径中的管道符号、大小于号以及与号，都不会被shell误解为特殊含义。所以*tar*命令会刚好得到5个参数，包括一个选项、一个打包后的文件名称以及三个要打包的目录。和下面这种带有安全隐患的写法比较一下：

注5：　或者在你编译Perl的时候找到的shell。不过实际情况中，在Unix类的系统上几乎总是*/bin/sh*。

注6：　CPAN上有一个shell到Perl的代码转换器，是在某个值得纪念的日子由Randal上传的。这个转换器用的就是这个技巧。

注7：　当然，如果你把它设置为$i = '$i'，也是能正常工作的。但没准哪天会有一个程序员“好心”帮你删掉那行！

注8：　或者按间接对象写法，第一个参数后面不用逗号，例如system { 'fred' } 'barney';。它实际上会运行程序barney，但却自己骗自己地说，程序名是'fred'。查看*perlfunc*文档了解更多信息。

注9：　参阅《Mastering Perl》一书当中有关安全的章节以获取更多详尽细节。

```
system "tar cvf $tarfile @dirs";  # 糟糕!
```

这样就会把一大堆压缩后的数据通过管道传送给*flintstone*命令，然后将它放在后台运行，同时把输出信息写入名为*betty*的文件。这还算好，要是@dirs里存放的是下面这样的内容，又会发生什么：

```
my @dirs = qw( ; rm -rf / );
```

@dirs是不是列表并不重要，Perl会简单地把它内插后变成单个参数传递给system。

这真是有点吓人 [注10]，尤其是当数据来自Web表单这样的用户输入的时候。所以也许你真的应该早些下决心，开始用system的多参数版本来启动子进程。当然这样你也会同时失去让shell帮你设定I/O重定向、后台进程等等事情的能力，天下没有免费的午餐。

另外请注意，虽然system的单一参数调用基本上和system的多参数调用非常相似：

```
system $command_line;
system '/bin/sh', '-c', $command_line;
```

但几乎没人会用后面这种写法，除非你想使用不同的shell来处理事情，比如C shell：

```
system '/bin/csh', '-fc', $command_line;
```

不过这种情形依然十分少见，因为即使“至尊（One True）” Shell [注11]也有足够的灵活性，尤其是在脚本处理方面。

system操作符的返回值是根据子进程的结束状态来决定的 [注12]。在Unix里，退出值0代表正常，非零退出值则代表有问题：

```
unless (system 'date') {
  # 返回0的话就代表执行成功
  print "We gave you a date, OK!\n";
}
```

注意绝大多数操作符遵从“真为正常，假为异常”的准则，而这里恰恰相反，所以若要

注10：除非你已经开始启用taint（污染）检查并做好十全准备，对用户传入的数据进行事先筛查，否则难免被某个用户戏弄。

注11：也就是*/bin/sh*或者任何在Unix系统中已经安装的最接近Bourne shell的那个。如果你没有安装“至尊”Shell的话，Perl会尝试随便调用一种命令行解释器，因此你得后果自负。请参考有关Perl移植的文档了解详细信息。

注12：其实是所谓“等待”状态，就是用子进程退出码乘以256，若出现崩溃内存转储则再加上128，还要加上触发程序终止的信号数号（如果有的话）。但我们很少需要解码分析，因为在绝大部分的应用中，简单的真/假值判断就够了。

套用“do this or die（做这件事，不然就得死）”的写法，我们就得先颠倒真假值。最简单的做法就是在system操作符之前加上逻辑操作符，也就是感叹号：

```
!system 'rm -rf files_to_delete' or die 'something went wrong';
```

注意在这个例子里面，若要显示错误信息不能引入$!变量，因为错误多半发生在*rm*命令的运行时刻，不是$!能捕获的系统调用相关错误。

环境变量

在使用这里讨论的任何方法启动其他进程的时候，可能会需要设置程序的环境。前面谈到我们可以在一个特定的工作目录下启动进程，然后它会从父进程继承这个目录。环境变量则是另一种常见的配置。

最广为人知的环境变量是PATH（若没有听说过，多半你在使用不支持环境变量的系统）。在Unix类的系统上，PATH是以冒号分隔的目录列表，其元素是可执行文件的搜索路径。当你键入*rm fred*这样的命令时，系统会在目录列表中依次寻找*rm*命令。Perl（或系统）会在需要时用PATH来搜索程序以便运行，运行之后该程序若需要调用其他程序，也会使用PATH来搜索它们。（当然如果指定了命令的全路径名，像*/bin/echo*，就没必要在PATH里搜索了。但这样写对大多数人来说太不方便了。）

在Perl中，环境变量可通过特殊的%ENV哈希取得，其中每个键都代表一个环境变量。在程序开始运行时，%ENV会保留从父进程（通常为shell）继承而来的设定值。修改此哈希就能改变环境变量，它会被新进程继承，也可以由Perl本身来使用。假如现在需要运行系统的*make*程序（进而运行其他程序），并且想以你的私有目录作为寻找命令（包括*make*自己）的首选位置，假如还要禁用IFS环境变量，以免*make*或其后的命令做出不正常的举动，就可以这么写：

```
$ENV{'PATH'} = "/home/rootbeer/bin:$ENV{'PATH'}";
delete $ENV{'IFS'};
my $make_result = system 'make';
```

新创建的进程一般会继承父进程的环境变量、当前工作目录、标准输入、输出与错误流和另外一些“小秘密”。可以参考系统与程序设计相关的文档来了解更多细节。但是请记住：修改从父进程继承的环境变量并不能影响shell或者其他父进程。

exec函数

到目前为止，我们提到的system函数的所有语法也都适用于exec函数。当然有一个重要的例外，system函数会创建子进程，子进程会在Perl睡眠期间执行任务。而exec函数却

导致Perl进程自己去执行任务。这类似于子程序调用与“goto”语句的差别。

例如，要运行*/tmp*目录下的*bedrock*命令并带上*-o args1*以及程序本身所调用的参数，可以这样写：

```
chdir '/tmp' or die "Cannot chdir /tmp: $!";
exec 'bedrock', '-o', 'args1', @ARGV;
```

当我们运行到exec时，Perl找到*bedrock*并且“跳进去”执行。此后，就没有Perl进程了[注13]，只有那个运行*bedrock*命令的进程。这样在*bedrock*运行结束时，没有Perl进程在等待。

为何要这样做呢？其实这个Perl程序的主要功能是为另一个程序的运行设定运行环境。你可以预先修改环境变量，修改当前工作目录，修改默认的文件句柄等等：

```
$ENV{PATH}  = '/bin:/usr/bin';
$ENV{DEBUG} = 1;
$ENV{ROCK}  = 'granite';

chdir '/Users/fred';
open STDOUT, '>', '/tmp/granite.out';

exec 'bedrock';
```

如果使用system而不是exec，Perl程序必须傻傻地等另一个程序运行完毕才能跟着收工，无疑这是在浪费系统资源。

话虽如此，实际上我们很少用到exec，一般都是将它和fork一起使用，这稍后会介绍。因此如果吃不准到底该用system还是exec，就总是用system好了，大多数情况下这都是稳妥的。

一旦要执行的命令启动后，Perl便放手退出，无法再控制它，因此在exec调用之后写的任何代码都无法运行，不过如果启动过程出现错误，那么后续的捕获语句还是可以继续运行的：

```
exec 'date';
die "date couldn't run: $!";
```

用反引号捕获输出结果

无论用system还是exec，所执行命令的输出都会送往Perl的标准输出。有时候我们感兴趣的是将输出结果捕获成字符串，以便后续进一步处理。要提取该输出信息，只要用反

注13： 其实还是那个老的进程，只不过执行了Unix的exec(2)或等效的系统调用。这样进程号是不变的。

引号代替单引号或双引号就可以了：

```
my $now = `date`;             # 捕获date命令的输出
print "The time is now $now"; # 换行符已经包含在捕获内容中
```

一般来说，*date*命令能输出长度约为30个字符的字符串，其中含有今天的日期与时间，最后接上一个换行符。当我们把*date*放在反引号里时，Perl会执行这个*date*命令并将其标准输出结果以字符串形式捕获。在这个例子中，字符串会被赋予$now变量。

这就像Unix shell的反引号一样，但shell还会做额外的处理：它会将最后一个换行符移除，让它易于与其他东西结合。Perl总是很诚实，它会直接使用接收到的真实输出。要在Perl中取得相同的结果，我们可以对取得的字符串进行一次chomp运算：

```
chomp(my $no_newline_now = `date`);
print "A moment ago, it was $no_newline_now, I think.\n";
```

反引号里面的内容就相当于单个参数形式的system函数调用[注14]并且会以双引号内的字符串的方式进行解释，换句话说，我们可以在里面使用反斜线转义和变量，它们会被适当地展开[注15]。比如要取得一系列Perl函数的说明文档，可以重复执行*perldoc*命令，每次使用不同的参数：

```
my @functions = qw{ int rand sleep length hex eof not exit sqrt umask };
my %about;

foreach (@functions) {
  $about{$_} = `perldoc -t -f $_`;
}
```

请注意，每次循环执行时$_的值都会不同，这让我们可以每次只改变一个参数就产生不同的命令并取得它的输出。另外，如果你还不熟悉这些函数，不妨趁此机会看一下说明文档，了解使用细节。

除了反引号，你还可以使用更为一般化的引起操作符qx()，它所完成的工作是一样的：

```
foreach (@functions) {
  $about{$_} = qx(perldoc -t -f $_);
}
```

和其他一般化的引起操作符类似，选用这种写法可以避免转义被引起内容中出现的分隔

注14： 就是说，它也总是以“至尊”Shell（*/bin/sh*）或相近的脚本解释器来解释的，就像system那样。

注15： 所以若要向shell发送一个真正的反斜线，就得连写两个反斜线。而在Windows环境中常常需要2个连续的反斜线，所以需要连写4个。

符。如果要执行的命令中本身就需要包含反引号，那就可以用qx()避免频繁转义带来的干扰。此外，选用一般化的引起操作符还有一个好处，就是如果选用单引号作为分隔符的话，可以禁止变量内插。比如希望选用shell的进程编号ID变量$$而不是Perl的时候，就可以用qx''避免Perl内插该变量：

```
my $output = qx'echo $$';
```

接下来要说明什么情况下不该使用反引号，但这个演示本身可能存在风险。我们的建议是，如果不需要捕获输出内容，就不要使用反引号[注16]，比如下面这个例子：

```
print "Starting the frobnitzigator:\n";
`frobnitz -enable`; # 请不要这么做!
print "Done!\n";
```

这里的问题是，就算你不需要，Perl也会尽力捕获该命令输出，然后直接丢弃。同时，这种形式的写法也让你失去了以system的多参数形式精确控制传入参数的能力。所以，在安全与效率的双重考虑之下，请改用system函数。

以反引号执行的命令的标准错误会继承Perl当前的标准错误输出。如果该命令将错误信息送到标准错误，就可能会显示在终端上，从而导致用户困惑，因为他并未运行*frobnitz*命令。如果你想要一并捕获标准输出和标准错误，就可以使用shell规范“将标准错误合并至标准输出”，在通常的Unix Shell中写成2>&1：

```
my $output_with_errors = `frobnitz -enable 2>&1`;
```

注意这会让标准错误与标准输出的信息交织在一起，就像在终端上看到的那样，当然可能因为缓冲的原因有顺序上的细微差别。如果你需要分别捕获标准输出和标准错误，就得考虑使用更加麻烦的解决方案[注17]。同样地，被执行的命令也会继承Perl当前使用的标准输入。我们日常执行的命令大都不会使用标准输入，所以没有这方面的问题。但是，如果*date*命令询问你要使用的时区（正如我们之前假设的），这样就会有问题，因为提示文字“which time zone”会被送至标准输出，成为被捕获内容的一部分，然后*date*会试着从标准输入读进数据。由于用户根本看不到提示文字，所以他不知道该输入数据！没多久，用户就会打电话给你，说你的程序卡住了。

因此，请勿使用会读取标准输入的命令。如果你不太确定它是否会从标准输入读取数据，请将标准输入重定向为从*/dev/null*读取数据，如下所示：

注16：　这其实就是所谓的“空”上下文。

注17：　比如使用标准Perl库里的IPC::Open3模块，或者自己编程处理派生子进程相关的事宜，稍后会有展示。

```
my $result = `some_questionable_command arg arg argh </dev/null`;
```

这样一来，子shell就会将输入重定向到*/dev/null*，接着再执行那个交互式命令。这样就算它要求输入，也只会读到文件结束符。

在列表上下文中使用反引号

如果命令会输出很多行，那么在标量上下文中使用反引号会得到一个很长的字符串，其中包含换行符[注18]。不过，如果是在列表上下文使用同样的反引号，则会返回输出字符串按行拆分的列表。

比如，Unix下的*who*命令会用多行文本列出当前登录到系统中的用户清单，每个人一行，如下所示：

```
merlyn     tty/42     Dec 7  19:41
rootbeer   console    Dec 2  14:15
rootbeer   tty/12     Dec 6  23:00
```

最左边的一列是用户名，中间列是TTY名（也就是登录到主机的终端连接的名称），其余的列则是登录日期与时间（也许还有远程登录信息，但本例没有）。在标量上下文中，我们会在一个变量中取得所有这些输出，所以必须自行拆开：

```
my $who_text = `who`;
my @who_lines = split /\n/, $who_text;
```

但在列表上下文中，则会自动取得拆成多行的数据：

```
my @who_lines = `who`;
```

现在@who_lines里会有多个拆分好的以换行符结尾的字符串。对这个结果调用chomp就可以删除所有元素末尾的换行符。不过不如换个思路，只要用foreach就可以逐行处理，将它们放在$_里：

```
foreach (`who`) {
  my($user, $tty, $date) = /(\S+)\s+(\S+)\s+(.*)/;
  $ttys{$user} .= "$tty at $date\n";
}
```

根据上面的数据，这个循环会迭代三次（你的系统可能有更多人登录，因此要循环的次数也会更多）。注意这里还用了正则表达式进行匹配，但并没有使用绑定操作符（=~），而是直接针对默认的$_进行匹配。这种写法干净利落，因为数据就在$_里。

注18：　实际上，计算机才不管什么换行不换行的，对它来说不过是一团字节序列。换行符是为了便于我们理解信息内容而加进去的，然后交代它对此做一些特殊处理。否则，换行符对计算机来讲和其他普通字符没什么两样。

我们注意到，这个正则表达式会寻找一个非空白的单词、数个空白、一个非空白的单词、数个空白，接着是剩余的所有单词，但是不包括换行符（因为点号默认不匹配换行符）[注19]。这种写法的另一个好处是，模式中每个捕获部分对应的就是$_里的数据。于是在循环处理第一行时，$1会是merlyn，$2会是tty/42，$3则是Dec719:41，一切自然而明晰。

不过，因为这个正则表达式是在列表上下文中进行运算的，所以如第八章所述，它不会像标量上下文中那样返回真假值，而是将被捕获的变量放进列表中。因此，$user最后会得到merlyn这个值，其他变量则依此类推。

循环中的第二条语句只是用来存储TTY与日期信息，之所以对哈希值（可能是undef）进行追加，是考虑到同一个用户可能有多次登录的情况（比如这个例子中的rootbeer）。

用IPC::System::Simple执行外部进程

运行外部命令或捕获其输出信息向来是件棘手的活儿，而Perl又意在各式平台无缝运行，这些平台又都有特别情况需要处理。Paul Fenwick的IPC::System::Simple模块把针对特定系统的复杂操作全都封装到幕后，并提供统一简洁的使用界面。目前它还不是Perl自带的模块，所以你得从CPAN下载安装[注20]。

这个模块使用起来真的非常简单，好像没什么可多说的。你可以直接用它提供的同名函数取代内置的system函数，但其幕后的处理更为健壮：

```
use IPC::System::Simple qw(system);

my $tarfile = 'something*wicked.tar';
my @dirs = qw(fred|flintstone <barney&rubble> betty );
system 'tar', 'cvf', $tarfile, @dirs;
```

它还提供了一个systemx函数，在执行外部命令时不会通过shell调用，所以不会碰到shell导致的意外状况：

```
systemx 'tar', 'cvf', $tarfile, @dirs;
```

如果要捕获外部命令的输出，只消把system或systemx改成capture或capturex就可以了，它们的作用就好像反引号一样（但更好些）：

注19：　现在你该明白为什么点号默认不会匹配换行符了吧。它可以让我们轻松写出上面这样的模式，而不必担心最后的换行符。另外请记住，如果用的是 Perl 5.12 或以上的版本，有一个 \N 表示除换行符之外的所有字符，意义明确，看起来也更漂亮。

注20：　地址为*http://search.cpan.org/dist/IPC-System-Simple*。

```
my @output = capturex 'tar', 'cvf', $tarfile, @dirs;
```

为了让这些函数能在Windows下正常运行，Paul做了大量工作。除了上面介绍的之外，这个模块还能做很多其他事情，具体内容还请参考该模块的文档。这里不深入介绍的原因是，某些功能涉及我们还没提过的引用的概念[注21]。如果你知道如何使用，我们建议你换掉内置的Perl函数而改用该模块提供的功能。

通过文件句柄执行外部进程

到目前为止，我们看到的方法都是由Perl同步控制子进程：启动一个命令，然后等待它结束，然后也许还会捕获其输出。但Perl其实也可以启动一个异步运行的子进程，并和它保持通信[注22]，直到子进程结束为止。

要启动并发运行的子进程，请将命令放在open调用的文件名部分，并且在它前面或后面加上竖线，也就是管道符号。因此,也有人将这种调用方式叫做“管道式打开（piped open）”。在两个参数的形式中，管道符号安放在要执行的命令的开头或者结尾：

```
open DATE, 'date|' or die "cannot pipe from date: $!";
open MAIL, '|mail merlyn' or die "cannot pipe to mail: $!";
```

第一个例子里的竖线在命令的右边，表示该命令执行时它的标准输出会连接到供程序读取的文件句柄DATE，就像在shell里执行*date|your_program*这个命令一样。在第二个例子里，竖线在左边，所以该命令的标准输入会连接到供程序写入的文件句柄MAIL，就像在shell里运行*your_program|mailmerlyn*这个命令一样。不论竖线在左在右，都会启动一个独立于Perl的进程[注23]。如果无法创建子进程，open就会失败。如果命令不存在或没有发生错误而正常结束，这在打开时通常不会有错误发生，但是在关闭时却会报错。稍后我们就会遇到这样的状况。

三个参数的形式看起来有点奇怪，因为就读取用的文件句柄而言，管道符号写在了命令“占位符”的后面。其实这里定义的是文件句柄打开模式，如果需要读取用的文件句柄，就用-|，如果需要写入用的文件句柄，就用|-，-的位置就好比是要执行的命令在管道传递中的位置：

注21：　不过，差不多已经到本书末尾了，而本系列的下一本书《Intermediat Perl》的开头讲的就是有关引用的知识。

注22：　通过管道或者操作系统提供的简易进程间通信机制。

注23：　如果Perl进程比命令早结束，默认情况下，等待中的读取数据的命令会得到文件结尾，而写入数据的命令在下一次写入时会收到“broken pipe”的错误信号。

```
open my $date_fh, '-|', 'date' or die "cannot pipe from date: $!";
open my $mail_fh, '|-', 'mail merlyn'
    or die "cannot pipe to mail: $!";
```

使用管道方式的open操作还可以使用三个以上的参数。第四个及其后的所有参数都将作为要执行的外部命令的参数，所以上面的写法可以拆成下面这种形式：

```
open my $mail_fh, '|-', 'mail', 'merlyn'
    or die "cannot pipe to mail: $!";
```

不管哪种写法，无论从哪点看，程序接下来的部分既不知道也不在乎，甚至得用尽办法才能分辨出这个文件句柄其实是连接到进程而非文件。所以，如果要从以读取模式打开的文件句柄读取数据，只要采用正常的文件读取方式即可：

```
my $now = <$date_fh>;
```

要想发送数据到mail进程（它此刻正在等待从标准输入读取发送给merlyn的邮件正文），只要利用输出到文件句柄的print就能完成：

```
print $mail_fh "The time is now $now"; # 假设$now以换行符结尾
```

总之，可以假想成这些文件句柄都连接了魔力文件，一个包含了*date*命令的输出，另一个可以自动用*mail*命令发送出去。

如果外部进程在连接到某个以读取模式打开的文件句柄后自行退出运行，那么这个文件句柄就会返回文件结尾标识符，就好像已经读完了正常的文件一样。当你关闭用来写入数据到某进程的文件句柄时，该进程会读到文件结尾标识符。所以，要结束邮件的发送，只要关闭这个文件句柄即可：

```
close $mail_fh;
die "mail: non-zero exit of $?" if $?;
```

关闭连接至进程的文件句柄会让Perl等待该进程结束以取得它的结束状态。结束状态会存入$?变量（联想到Bourne Shell里的同名变量了吗？），它的值就与system函数返回的数值一样：零表示成功，非零值代表失败。每个结束的进程都会覆盖掉前一个返回值，所以，如果你需要这个值，请尽快保存。（如果你好奇的话，$?变量也会存储前一次system或用反引号圈引的命令的返回值。）

这些进程间的同步方式，就像shell中被管道连接的命令一样。如果你试着想要读取数据，但是没有任何数据送达，程序就会暂停（但不会消耗额外的CPU时间），直到送出数据的进程有数据发送为止。同样，如果产生数据的进程超出读取进程的速度，它就会减速运行，直到读取数据的进程赶上为止。进程之间会有缓冲区（一般是8KB大小），这样可以避免它们相互锁定。

为什么要用文件句柄的方式来和进程打交道呢？嗯，假如要根据计算的结果来决定写到其他进程的数据，这是唯一简单的做法。可是如果只想读取，除非想在结果出现时立刻取得，否则反引号通常更易于使用。然而如果子进程不时有数据要送给父进程的话，就必须用管道了。

比如Unix的*find*命令可以依照文件属性来寻找文件位置。但如果文件数目很多的话，这会耗费不少时间，尤其是可能要从根目录开始找时。虽然可以将*find*命令放在反引号内，但你也可以在每找到一个文件时就立即取得它的名称。这通常是比较好的做法：

```
open my $find_fh, '-|',
  'find', qw( / -atime +90 -size +1000 -print )
     or die "fork: $!";
while (<$find_fh>) {
  chomp;
  printf "%s size %dK last accessed %.2f days ago\n",
    $_, (1023 + -s $_)/1024, -A $_;
}
```

这里的*find*命令是要查找那些90天内未被访问过的，占用空间超过1 000个块的大文件，它们非常适合被归档到永久性存储介质中。在*find*工作时，Perl会等待。每找到一个文件，Perl会对每个传进来的文件名作出响应并进一步显示文件的相关信息供分析。如果我们用反引号来写这个程序的话，就得等到*find*彻底搜索完才能看到第一行输出。从任务监控角度来说，往往看到执行的最新进展才能让人放心。

用fork进行深入和复杂的工作

除了之前介绍的高级接口外，Perl还提供了近乎直接执行Unix及某些其他系统的低级进程管理系统调用的能力。如果你从来没有这样做过[注24]，不妨略过本节。虽然本章的篇幅不足以详述全部细节，但我们至少先来看看下面这条语句的低级实现：

```
system 'date';
```

如果换用低级系统调用，大致可以写成：

```
defined(my $pid = fork) or die "Cannot fork: $!";
unless ($pid) {
  # 能运行到这里的是子进程
  exec 'date';
  die "cannot exec date: $!";
}
```

注24： 或者你运行在一个不支持fork的系统上。但是Perl开发人员正努力在此系统上实现支持fork，即使系统的底层进程模型和Unix上的差距非常大。

```
# 能运行到这里的是父进程
waitpid($pid, 0);
```

这里检查了fork的返回值，它在失败时会返回undef。如果成功了，则下一行开始就会有两个不同的进程在运行。因为只有父进程的$pid不是零，所以只有子进程会执行exec函数。父进程会略过该部分，直接执行waitpid函数，也就是在那里等待特定的子进程结束（在这期间，若是其他的子进程结束执行，则会被忽略掉）。如果这些听起来像天书的话，那也没关系，继续使用system函数。你不会被朋友耻笑的。

在付出额外的代价之后，程序员也获得了完整的控制权，可以创建任意管道、对文件句柄进行重新安排，也可以进一步了解子进程ID和父进程ID。但对于这一章来说，这些细节还是太复杂了，想要深入的话，请参阅*perlipc*文档，或者阅读详细谈论系统编程的书籍。

发送及接收信号

Unix信号[注25]是发送给进程的一条简短的消息。信号无法作详尽说明，它就像汽车喇叭声一样：喇叭声对你而言，可能代表"小心！桥断了"、"绿灯啦！快走"、"快停下！车顶上有个小孩"，或是"Hello, world"。好在，Unix信号比这些稍好认一些，因为针对不同的情况会有不同的信号[注26]。信号会用不同的名称相互区分（像SIGINT就代表中断信号），另外还有一个相关的小整数也可用来识别（它的取值范围从1到16、1到32或是1到63，依你的Unix系统而定）。通常某个重大事件发生时就会发出信号，比如在终端上按下Control+C这样的中断字符，就会给与此终端相连的所有进程发送SIGINT信号[注27]。某些信号可能是由系统自动发送的，但也可能来自别的进程。

可以从Perl进程发送信号给别的进程，但是得先知道目标进程的ID。要说明如何取得进程ID可能有点复杂[注28]，不过假设你已经知道要发送SIGINT信号给进程4201，而且我们知道SIGINT对应的编号是2，那么做法简单明了[注29]：

注25：Windows没有信号的概念，它是一头完全不同于Unix的怪兽。

注26：当然与以上这些状况并非完全相似，不过确实有相近的Unix信号。比如信号SIGHUP、SIGCONT、SIGINT以及假的SIGZERO（信号编号0）。

注27：你可能以为按下Control+C就会中止程序，其实这样做只是送出一个SIGINT信号，而这个信号在默认情况下恰巧能中止程序而已。稍后就能看到如何在收到SIGINT信号时做些其他事，而不单单是中止程序运行。

注28：进程ID要么是你自己以fork产生的，要么是从某个文件中或是从外部程序中取得。从外部程序中取得进程ID更困难，而且更容易出错，因此那些要运行很久的程序往往把自己的进程ID写在一个文件里面，并且通常在程序文档中对此文件明确说明。

注29：在Unix系统上，要列出所有信号的数字代号，可以在命令行执行*kill -l*命令查看。

```
kill 2, 4201 or die "Cannot signal 4201 with SIGINT: $!";
```

发送信号的命令取名为“kill”，因为发明信号的主要目的之一就是中止运行了太久的进程。你也可以用字符串'INT'代替2发送信号，所以无需记住信号编号：

```
kill 'INT', 4201 or die "Cannot signal 4201 with SIGINT: $!";
```

你还可以使用=>操作符，这样信号名称就会自动作为裸字字符串：

```
kill INT => 4201 or die "Cannot signal 4201 with SIGINT: $!";
```

如果进程早已退出[注30]，就会收到表示失败的返回值。所以我们可以借用这个技巧来判断某个进程是否仍然存活。编号为0的特殊信号的意思是："看看我能否向它发送信号，但目前并不是要真的发送，所以请不要叨扰它。"因此，探测进程存活与否的代码可以写成这样：

```
unless (kill 0, $pid) {
  warn "$pid has gone away!";
}
```

接收信号好像比发送信号要有趣些。你为什么会想要这么做呢？假设你有一个程序会在*/tmp*目录里创建文件。正常情况下，程序结束前就会删除这些文件。如果有人在程序运行时按下Control+C，结果就会在*/tmp*目录里留下垃圾，而这是很不礼貌的事情。要解决这个问题，可以创建一个负责清理的信号处理程序：

```
my $temp_directory = "/tmp/myprog.$$"; # 在这个目录下创建文件
mkdir $temp_directory, 0700 or die "Cannot create $temp_directory: $!";

sub clean_up {
  unlink glob "$temp_directory/*";
  rmdir $temp_directory;
}

sub my_int_handler {
  &clean_up();
  die "interrupted, exiting...\n";
}

$SIG{'INT'} = 'my_int_handler';
.
.   # 时间流逝，程序在运行着，在临时目录中创建一些
.   # 临时文件，然后可能有人按了 Control+C
.
```

注30：　如果你不是超级用户或者该进程是别人的，那么发送信号就会失败。无论如何，给别人的进程发送SIGINT也总是很失礼的举动。

```
# 这里是正常执行流程的结尾部分了
&clean_up();
```

对特殊哈希%SIG赋值会启动信号处理程序（直到撤销为止）。哈希键是信号名称（注意，不用写固定的SIG前缀），哈希值是[注31]子程序名（注意，不需要与号）。现在只要收到SIGINT信号，Perl就会暂停手上的事务并立刻执行信号处理子程序。这里的子程序会清理临时文件并退出。当然，即使没有按Control+C，也还是会在正常执行结束之前调用&clean_up()。

假如信号处理子程序没有结束程序而是直接返回，那么程序会从先前中断的地方继续运行。如果该信号只是要中断某些事情，而不是停止整个程序的话，这种时候就该用返回而非退出。举例来说，假设处理文件里的每行都慢到要花几秒钟的时间，而你想要在收到信号时停止处理，却不想让进行中的这一行中断，这时，只要在信号处理子程序中设定一个标记，然后在每行处理结束时检查它即可：

```
my $int_count = 0;
sub my_int_handler { $int_count++ }
$SIG{'INT'} = 'my_int_handler';
...;
while (<SOMEFILE>) {
  ...; # 一些需要花几秒时间来执行的操作 ...
  if ($int_count) {
    # 看到进来的中断信号了!
    print "[processing interrupted...]\n";
    last;
  }
}
```

这样在处理完每行之后，如果没有按下Control+C，$int_count的值将会是0，循环会继续下一次的处理。但是如果发生中断，则中断处理程序会将$int_count加1，之后的检查会让循环中止。

所以，你可以设定标记或是直接跳离程序，而这两种方法足以涵盖绝大部分捕获信号的需求。对大部分情况来说，Perl要一直等到安全妥当的时机，才会动手处理进来的信号。比如Perl在分配内存和调整内部数据结构的阶段是不会理睬处理大多数的信号的[注32]。但Perl会立即处理SIGILL、SIGBUS以及SIGSEGV这些信号，所以对程序运行而言，这些信号可能会造成不安全。

注31： 哈希值也可以是子程序引用（这应该是比较恰当的做法），但本书不会谈及有关引用的话题。

注32： 如果你真的关心Perl是如何处置的，请参阅*perlipc*文档。

习题

以下习题答案参见第339页上的“第十六章习题解答”一节：

1. [6]写一个程序，让它进入某个特定（在程序内写死）的目录，比如系统根目录。然后执行*ls -l*命令获得该目录内的长格式目录列表。（如果你使用非Unix的系统，请使用该系统上相应的命令来取得详细的目录列表。）

2. [10]修改前面的程序，让它将命令的输出送到当前目录下的*ls.out*文件，错误输出则送到*ls.err*文件。（请不必对结果文件为空的情况做任何特别处理。）

3. [8]写一个程序，用它解析*date*命令的输出并判断今天是星期几。如果是工作日，输出`get to work`，否则输出`go play`。*date*命令的输出中，星期一是用`Mon`来表示的[注33]。如果你使用非Unix系统因而没有*date*命令，那就做一个假的小程序，只要输出像*date*命令的输出结果即可。如果你保证不问下面两行小程序的原理，我们就无偿奉上：

   ```
   #!/usr/bin/perl
   print localtime( ) . "\n";
   ```

4. [15]（仅限于Unix系统）写一个无限循环程序，让它能捕获信号并报告之前收到过该信号的次数。如果收到`INT`信号就退出程序。如果可以在命令行使用*kill*命令，可以像下面这样发送特定信号：

   ```
   $ kill -USR1 12345
   ```

   如果你没法使用命令行工具*kill*，那就写一个辅助程序发送信号。实际上用Perl的单行程序就能做到：

   ```
   $ perl -e 'kill HUP => 12345'
   ```

注33： 起码在英语环境中是这样的。你可以根据你的系统设定灵活调整。

第十七章

高级Perl技巧

到目前为止，本书已经展现了Perl语言的核心，也就是每个Perl用户都需要明白的东西。然而也还有些可选的技术，它们并非必知必会，不过仍足以成为你的工具箱里的珍贵工具。我们把它们当中最重要的那些都收编在这一章里。读完本章后，你差不多就可以开始阅读《Intermediate Perl》，进入Perl学习的下一个阶段。

当然别被这一章的标题吓到了，这些技术并非特别难懂。我们所谓的高级，其实只是从初学者的角度来说的。所以第一次阅读本书的时候，为了尽快使用Perl，你可能想要跳过这个章节，在一两个月之后，为了更多了解Perl，你也可以回头再学。就把这一章想像为超级脚注好了[注1]。

切片

我们往往只需要处理列表当中的少量元素。假设Bedrock图书馆用一个大文件来存放读者信息[注2]，文件中的每一行都描述了一个读者，用6个字段（冒号作为分隔符）分别描述读者姓名、借书证号码、住址、家庭电话、工作电话和当前借阅数量。文件内容类似于：

```
fred flintstone:2168:301 Cobblestone Way:555-1212:555-2121:3
barney rubble:709918:3128 Granite Blvd:555-3333:555-3438:0
```

图书馆的某个应用程序只需要借书证号码和借阅数量，不关心其他数据。所以可以这样来读出需要的两个字段：

注1：　我们真的试图把这种超级脚注加入初稿，但被O'Reilly的编辑断然拒绝。

注2：　虽然其实他们也知道应该用功能齐全的数据库而非平面文件(flat file)，他们计划好了，下个冰河期一到，就立马升级系统。

```
while (<$fh>) {
  chomp;
  my @items = split /:/;
  my($card_num, $count) = ($items[1], $items[5]);
  ...  # 现在可以用这两个变量来继续工作
}
```

但@items数组不会有其他的用处，看来是一种浪费[注3]。也许用一组标量来容纳split的结果会更好些：

```
my($name, $card_num, $addr, $home, $work, $count) = split /:/;
```

好的，这确实避免了引入导致浪费的数组@items，但我们现在又多出来4个不需要的标量。有人图方便，将这种占位变量命名为$dummy_1，表示他们并不关心这些从split所取得的元素。但Larry觉得这么做太麻烦，因此他引入了一种特殊的undef写法。如果被赋值的列表中含有undef的话，就干脆忽略源列表的相应元素：

```
my(undef, $card_num, undef, undef, undef, $count) = split /:/;
```

这不是更好么？应该说这样确实避免了引入不需要的变量，但问题是要弄清undef的数量，才能正确获取$count。而且如果列表元素数量稍微多些就更麻烦了。若要获取stat结果中的mtime值则必须写出如下的代码：

```
my(undef, undef, undef, undef, undef, undef, undef,
  undef, undef, $mtime) = stat $some_file;
```

如果弄错了undef的数量，就可能获得atime或者ctime的值，这会是一个很难发现的bug。更好的办法是：Perl可以把列表当作数组，用索引取得里面的值。这就是所谓的列表切片（*list slice*）。本例中，因为mtime是stat返回的列表的9号元素[注4]，我们可以通过下标数字获取数据：

```
my $mtime = (stat $some_file)[9];
```

这里stat周围的括号是必须的，因为需要用它们产生列表上下文。如果你像下面这样写，就不会正常工作：

```
my $mtime = stat($some_file)[9];  # 语法错误!
```

注3：其实也并非很大的浪费，但请先听我们的。那些不明白切片的程序员仍会使用这些技术，因此当你需要处理别人的代码时，在此全部看一遍还是值得的。

注4：其实是第10个元素，但是索引号是数字9，因为第一个元素的索引号是0。这和数组索引号从零开始是一样的。

列表切片必须在一对圆括号引起的列表后面有一个由方括号括起的下标表达式。但是函数为引入参数而使用的圆括号不算。

回到Bedrock图书馆的例子，需要处理的列表是split的返回值。我们可以用切片与下标取出元素1和元素5：

```
my $card_num = (split /:/)[1];
my $count = (split /:/)[5];
```

像这样的标量上下文切片（每次取一个列表元素）用起来也不错，但是如果能避免两次调用split，就会更加简单和高效。因此让我们在列表上下文中使用列表切片一次取得两个值：

```
my($card_num, $count) = (split /:/)[1, 5];
```

上面的索引会从列表里取出元素1和元素5，将其返回为一个两个元素的列表。然后我们把结果赋值到两个my变量，这恰好是我们所期望的：一次切片成型，并轻松对两个变量赋值。

切片常常是从列表中读取少量数据的最简单方法。下面的例子中，我们从列表中取出第一个和最后一个元素，借助索引-1代表最后一个元素这一事实[注5]：

```
my($first, $last) = (sort @names)[0, -1];
```

切片的下标可以是任意顺序的，也可以重复。下面这个例子从10个元素的列表中找出5个元素：

```
my @names = qw{ zero one two three four five six seven eight nine };
my @numbers = ( @names )[ 9, 0, 2, 1, 0 ];
print "Bedrock @numbers\n";  # 打印：Bedrock nine zero two one zero
```

数组切片

前面的例子还可以进一步简化。在从数组（而不是从列表）切出元素时，圆括号并非必需的。所以我们可以这样进行切片：

```
my @numbers = @names[ 9, 0, 2, 1, 0 ];
```

这不单是省略圆括号，其实是一种访问数组元素的不同写法：数组切片(*array slice*)。我们曾经在第三章中提到@names前面的@符号表示“所有元素”。其实从语言学角度来看，

注5：　用排序方法找最大／最小值并非最有效的，但是Perl的排序还是非常高效的，对元素数量不超过三位数的列表来说是可以接受的。

它更像是一种复数标志，非常类似英文“cats”和“dogs”后面的字母“s”。在Perl中$符号意味着一个东西，而@符号意味着一组东西。

切片总是一个列表，所以数组切片总是使用一个@符号来标识。当你看见类似@names[...]之类的写法时，需要以Perl的习惯来看开头的符号和结尾的方括号。方括号意味着你要检索数组成员，@符号则意味着获取的是整个列表[注6]，而$符号意味着获取单个元素。请参考图17-1。

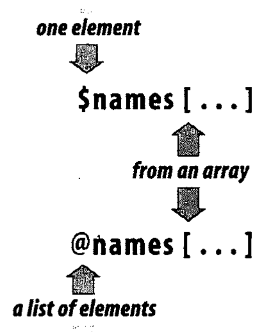


图17-1：数组切片和单个元素的区别

变量引用之前的标点符号（$或@符号）决定了下标表达式的上下文。如果前面有个$符号，下标表达式就会在标量上下文中求得索引值。但如果之前有个@符号的话，下标表达式就会在列表上下文中计算，从而得到索引列表。

所以这里看到的@names[2,5]和($names[2],$names[5])代表同样的列表。因此如果你希望得到值列表，就可以用数组切片的写法。任何需要列表的地方都可以替换成更简单的数组切片。

但有一个切片可以工作，列表却不能的场合。那就是切片可以被直接内插到字符串中去：

```
my @names = qw{ zero one two three four five six seven eight nine };
print "Bedrock @names[ 9, 0, 2, 1, 0 ]\n";
```

如果我们想要内插@names，就会得到数组所有成员构成的字符串，元素之间用空格隔开。如果我们要内插的是@names[9,0,2,1,0]，就会得到指定数组元素构成的字符串，同

注6：　当然我们说到整个列表的时候，并不是意味着必须要有多于一个的元素，毕竟列表也可以为空。

样用空格隔开[注7]。让我们回到Bedrock图书馆的例子，假设我们的程序需要修改读者Slate先生的地址和电话号码，因为他刚刚搬到了Hollyrock山庄的某间大房子。如果我们得到了一个存有关于他的信息列表的@items，那么就可以按如下方式简单地修改数组中的那两个元素：

```
my $new_home_phone = "555-6099";
my $new_address = "99380 Red Rock West";
@items[2, 3] = ($new_address, $new_home_phone);
```

和前面一样，数组切片可以用更简洁的方式来表示一系列元素。在这个例子里，最后一行程序代码其实就是赋值给($items[2],$items[3])，但更简洁高效。

哈希切片

和数组切片相似，也可以用哈希切片（*hash slice*）的方式从哈希里切出一些元素。还记得三个选手的保龄球积分么？它们存放在哈希%score中。我们可以用哈希元素所构成的列表来取出这些积分，或是使用切片。这两个技巧实际的效果相当，但第二种做法更加简洁高效：

```
my @three_scores = ($score{"barney"}, $score{"fred"}, $score{"dino"});

my @three_scores = @score{ qw/ barney fred dino/ };
```

切片一定是列表，因此哈希切片也是用@符号来表示。不管你是否认为我们罗嗦，我们就是想强调哈希切片和数组切片是类似的。如果Perl程序中看到类似@score{...}这样的写法，你需要以Perl的习惯来看开头的符号和结尾的花括号。花括号意味着你要检索哈希成员，@符号意味着获取的是整个列表，而美元符号代表的是单个元素。请参考图17-2。

如同我们在数组切片中所见的，变量前置的标点符号（可能是美元符号或@符号）决定了下标表达式的上下文。如果前置美元符号，下标表达式就会在标量上下文中运算并返回单一键值[注8]。但如果前置@符号，下标表达式就会在列表上下文中运算并返回一个键值列表。

注7：更准确地说，列表的这些元素会被Perl内置的$"变量的内容填充，而它的默认值就是空格。这一般不应改变。在内插时，Perl实际上会执行join $",@list，而@list就代表那个列表表达式。

注8：有一个例外，但很难碰到，因为如今的Perl程序中很少使用它。参见*perlvar*文档中关于$;的部分。

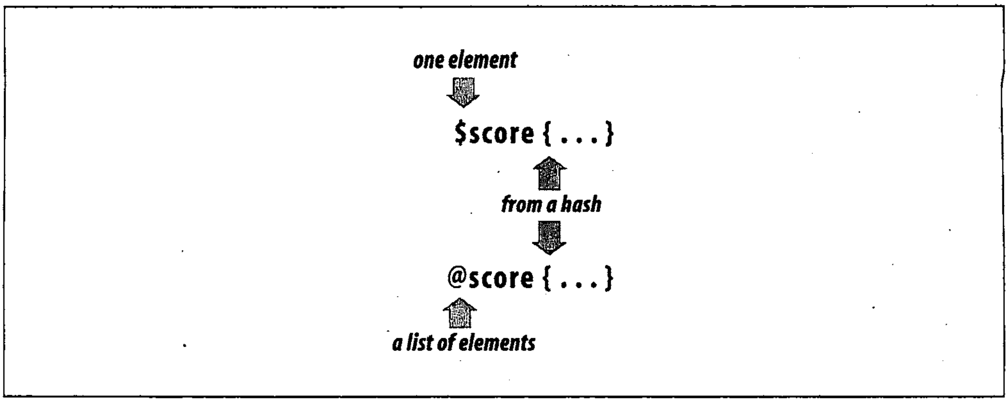


图17-2：哈希切片和单个元素的区别

这里自然会有人问，为什么提到哈希时并没有用百分比符号（%）？因为百分比符号表示整个哈希，哈希切片（就像其他切片）一定是列表而不是哈希[注9]。在Perl中，美元符号代表单个东西，但@符号则代表一组东西，而百分比符号代表一整个哈希。

如同我们在数组切片中所见的，在Perl的任何地方，你都可以使用哈希切片来代表哈希里相应的元素。因此下面的程序可以将这些选手的保龄球分数存入哈希，不必担心意外修改哈希中的其他元素：

```
my @players = qw/ barney fred dino /;
my @bowling_scores = (195, 205, 30);
@score{ @players } = @bowling_scores;
```

最后一行所做的事相当于对($score{"barney"},$score{"fred"},$score{"dino"})这个具有三个元素的列表进行赋值。

哈希切片也可以被内插进字符串。下面的例子会输出我们喜欢的保龄球选手的积分：

```
print "Tonight's players were: @players\n";
print "Their scores were: @score{@players}\n";
```

捕获错误

程序写出来往往会碰到各种意外，甚至无法正常工作，我们希望能在程序停止运行前知道究竟是什么原因导致的。其实，对错误的处理是编程工作中不可或缺的重头戏，虽然有关这方面我们可以写上整整一本书，不过这里作为入门，还是先作一些粗略的介绍。如果有兴趣，可以阅读本系列的第三本书《Mastering Perl》，我们会对Perl里面的错误处理作更为深入的阐释。

注9：　哈希切片是一个切片而非哈希，如同炉火是指火而非炉，而火炉是指炉而非火。

用eval

有时看上去平淡无奇的代码却能在程序中导致严重错误。以下这些比较典型的错误语句都能让程序崩溃：

```
my $barney = $fred / $dino;       # 除零错误？

my $wilma = '[abc';
print "match\n" if /\A($wilma)/;  # 非法的正则表达式错误？

open my $caveman, '<', $fred      # 用户提供的数据错误？
  or die "Can't open file '$fred' for input: $!";
```

你可能会制造一些状况或找出其中某些错误，然而这种方式很难找出所有错误。比如上面的例子中，你要如何检查字符串$wilma才能确保它引入的正则表达式合法呢[注10]？好在Perl提供了简单的方式来捕获代码运行时可能出现的严重错误，即把代码包裹在eval块里：

```
eval { $barney = $fred / $dino };
```

现在即使$dino是零，这一行代码也不至于让程序崩溃。只要eval发现在它的监察范围内出现致命错误，就会立即停止运行整个块，退出后继续运行其余的代码。注意，eval块的末尾有一个分号。实际上，eval只是一个表达式，而不是类似于while或foreach那样的控制结构。所以在要监察的语句块末尾必须写上分号。

eval的返回值就是语句块中最后一条表达式的执行结果，这一点和子程序相同。所以，我们可以把语句块最后的$barney从eval里拿出来，将eval的运行结果赋值给它。这样，$barney变量的声明就位于eval外部，便于后续使用：

```
my $barney = eval { $fred / $dino }
```

如果eval捕获到了错误，那么整个语句块将返回undef。所以，你可以利用定义或操作符对最终的变量设定默认值，比如NaN（表示"Not a Number"，非数字）：

```
use 5.010;
my $barney = eval { $fred / $dino } // 'NaN';
```

当运行eval块的期间出现致命错误时，停下来的只是这个语句块，整个程序不会崩溃。

当eval结束时，我们需要判断它是否正常退出或是否捕获到严重错误。如果捕获到错误，eval会返回undef，并且在特殊变量$@中设置错误消息，比如：Illegal division by zeroat my_program line 12。如果没有错误发生，$@就是空的。当然，这时候通过检查$@取值的真假就可以判断是否有错误发生。所以，我们常常会看到eval语句块之后

注10：　要检查正则表达式是否合法其实很简单，不过我们还没介绍相关工具。请参阅《Intermediate Perl》一书中关于正则表达式对象的章节。

立即跟上这样一段检测代码：

```
use 5.010;
my $barney = eval { $fred / $dino } // 'NaN';
print "I couldn't divide by \$dino: $@" if $@;;
```

也可以通过检查返回值来判断，只要正常工作时能返回真值就可以了。不过，其实你最好用下面这种写法，如果可以的话：

```
unless( eval { $fred / $dino } ) {
    print "I couldn't divide by \$dino: $@" if $@;
}
```

有时候你想测试的部分即使成功也并没有什么有意义的返回值，所以得另外构造一个有返回值的代码块。如果eval捕获到了错误，就不会执行最后一条语句，本例中也就是1：

```
unless( eval { some_sub(); 1 } ) {
    print "I couldn't divide by \$dino: $@" if $@;
}
```

在列表上下文中，捕获到错误的eval会返回空列表。下面这行中，如果eval失败的话，@averages最终只会得到两个元素，因为eval返回的是空列表，等于不存在：

```
my @averages = ( 2/3, eval { $fred / $dino }, 22/7 );
```

eval块和其他语句块一样，所以可以设定词法(my)变量的新作用域，并且块内的语句数目不限，只要你需要，多少都可以。比如下面这个eval块对多处潜在的致命错误作了事先防范：

```
foreach my $person (qw/ fred wilma betty barney dino pebbles /) {
  eval {
    open my $fh, '<', $person
      or die "Can't open file '$person': $!";

    my($total, $count);

    while (<$fh>) {
      $total += $_;
      $count++;
    }

    my $average = $total/$count;
    print "Average for file $person was $average\n";

    &do_something($person, $average);
  };

  if ($@) {
    print "An error occurred ($@), continuing\n";
  }
}
```

这段程序中的eval能捕获多少错误？如果打开文件时出错，你会捉到它。计算平均值的时候如果除以零，也会被捉到，不会使你的程序过早结束。eval还保护对神秘的名为&do_something的子程序的调用，不管里面在做什么，只要出现致命错误，一样会捉到。这个功能非常贴心，我们常常会用到其他人写的子程序，对于内部细节完全没有概念，但又不想因为它而导致程序崩溃。有些人故意使用die抛出错误消息，因为他们期待外部使用者通过eval捕获错误并做适当处理。这部分内容我们马上就会介绍。

如果在处理foreach提供的列表中的某个文件时发生错误，你会得到错误消息，但程序会继续处理下一个文件而没有更多的抱怨。

你还可以把eval块写在另一个eval块里面嵌套，Perl不会被搞糊涂的。内层的eval负责捕获它自己块中的错误，不会把错误泄露到外层块中。当然，在内层eval执行完毕后，如果它捉到错误并用die报告的话，外层的eval就会捉到这个错误报告。我们可以修改上面的代码，把除零错误放在内层块中单独捕获：

```
foreach my $person (qw/ fred wilma betty barney dino pebbles /) {
  eval {
    open my $fh, '<', $person
      or die "Can't open file '$person': $!";

    my($total, $count);

    while (<$fh>) {
      $total += $_;

      $count++;
    }

    my $average = eval { $total/$count } // 'NaN'; # 内层 eval
    print "Average for file $person was $average\n";

    &do_something($person, $average);
  };

  if ($@) {
    print "An error occurred ($@), continuing\n";
  }
}
```

总共有4种类型的错误是eval无法捕获的。第一种是出现在源代码中的语法错误，比如没有匹配的引号，忘写分号，漏写操作符，或者非法的正则表达式等等：

```
eval {
  print "There is a mismatched quote';
  my $sum = 42 +;
  /[abc/
  print "Final output\n";
  }
```

*perl*解释器的编译器会在解析源代码时捕获这类错误并在运行程序前停下来。而`eval`仅仅能捕获Perl运行时出现的错误。

第二种是让*perl*解释器本身崩溃的错误，比如内存溢出或者收到无法接管的信号。这类错误会让*perl*解释器意外终止运行，既然它已经退出运行了，自然无法用`eval`捕获。如果对此感兴趣，请参阅*perldiag*文档中前面有(X)代码的错误清单。

第三种`eval`块无法捕获的错误是警告，不管是由用户发出的（通过`warn`函数），还是Perl自己内部发出的（通过打开`-w`这个命令行选项，或者使用`use warnings`编译指令）。要让`eval`捕获警告有专门的一套机制，请参考Perl文档中`__WARN__`伪信号相关的内容。

最后一种错误其实也不算是错误，不过放在这里提一下比较好。`exit`操作符会立即终止程序运行，就算你从`eval`块内的子程序来调用它。我们每次调用`exit`的时候，就是希望它立即让程序停止运行，这可没什么好多考虑的，自然`eval`也不会阻止你这样做。

这里，我们还要提醒一下使用`eval`不当而造成危险的情况。实际上，你经常会听别人说，为了安全不要在程序里用`eval`。从某种意义上说，他们的意见是正确的，只有在相当关注安全时，才应该使用`eval`。不过其实他们所说的只是`eval`语句的另一种用法，即称为“`eval`字符串”的用法。这种形式的`eval`会把拿来的字符串直接当作Perl源代码编译，然后执行，这就好比你手工在程序里面敲上这段代码。请看下面的例子，任何出现在字符串中的东西都会被当作Perl代码来解释执行：

```
my $operator = 'unlink';
eval "$operator \@files;";
```

如果关键字`eval`后面紧跟的是花括号围起来的代码块，正如本节绝大多数例子那样，就无需担心——那是安全的`eval`用法。

更为高级的错误处理

每种语言都有自己的一套处理错误的方式，不过普遍都有一个叫做异常（*exception*）的概念。具体来说，就是尝试运行某段程序，如果出现错误就抛出（throw）异常，然后等待后续负责捕获（catch）这类异常的代码做相应处理。在Perl里面最基本的做法是，用`die`抛出异常，然后用`eval`捕获异常。我们可以通过识别保存在`$@`里面的错误消息判断究竟出了什么问题：

```
eval {
  ...;
  die "An unexpected exception message" if $unexpected;
  die "Bad denominator" if $dino == 0;
  $barney = $fred / $dino;
  }
```

```
if ( $@ =~ /unexpected/ ) {
  ...;
  }
elsif( $@ =~ /denominator/ ) {
  ...;
  }
```

不过这类代码有诸多弊端，最明显的就是$@变量的动态作用域问题了。简单来说，由于$@是一个特殊变量，而你所写的eval也许会被包含在另一个高层的eval里面（很有可能你对此一无所知），那就需要确保这里出现的错误不干扰高层出现的错误：

```
{
local $@; # 不干扰高层错误

eval {
  ...;
  die "An unexpected exception message" if $unexpected;
  die "Bad denominator" if $dino == 0;
  $barney = $fred / $dino;
  }
if ( $@ =~ /unexpected/ ) {
  ...;
  }
elsif( $@ =~ /denominator/ ) {
  ...;
  }
}
```

这还不是全部，像这样比较微妙的问题很容易导致错误。Try::Tiny模块帮你解决大部分这类问题（如果你想了解到底有哪些问题，请参阅它的文档）。该模块并未包含在标准模块库当中，所以你得自己从CPAN下载安装[注1]。最基本的用法如下所示：

```
use Try::Tiny;

try {
  ...; # 某些可能会抛出异常的代码
  }
catch {
  ...; # 某些处理异常的代码
  }
finally {
  ...;
  }
```

这里try的作用就好比是之前看到的eval语句。只有当出现错误时，才会运行该结构中catch块的部分。并且，不管是否出错，最终都会运行finally块中的内容，以便实施清理工作。其实我们也可以省略catch或finally块，单单用try直接忽略出现的错误：

注1： 请访问*http://search.cpan.org/dist/Try-Tiny*。

```
my $barney = try { $fred / $dino };
```

我们可以用catch处理错误。为了避免干扰$@，Try::Tiny把错误消息放到了默认变量$_里。当然，你还是能够访问$@的，只不过我们用Try::Tiny的目的之一就是要规避$@带来的潜在干扰：

```
use 5.010;

my $barney =
  try { $fred / $dino }
  catch {
    say "Error was $_"; # 不用$@
    };
```

不管有错没错，都会运行finally块里的代码。如果@_中有参数内容，就说明之前出现错误：

```
use 5.010;

my $barney =
  try { $fred / $dino }
  catch {
    say "Error was $_"; # 不用 $@
    }
  finally {
    say @_ ? 'There was an error' : 'Everything worked';
    };
```

autodie

从5.10.1版起，Perl自带autodie编译指令，我们可以在程序中让它负责自动抛出异常。本书绝大部分示例中，都是在open打开文件句柄时用die检查是否发生错误：

```
open my $fh, '>', $filename or
  die "Couldn't open $filename for writing: $!";
```

它本身看起来并没问题，不过，我们真的需要每次用open的时候都反复检查吗？其他和操作系统交互的内置函数要是也失败了呢？其实，我们不用每次都麻烦地写上一句“or die...”，让autodie自动帮你加上去好了：

```
use autodie;

open my $fh, '>', $filename; # 仍旧会在错误发生时调用die函数
```

如果打开操作失败，你会得到一条错误消息，内容正如我们原来构造的那样：

```
Can't open '/does/not/exist' for writing: 'No such file or directory'
```

autodie模块默认会对一系列Perl内置的用于处理文件、文件句柄、进程间通信和套接字的函数自动施行监管。不过你也可以自行指定需要施行autodie监管的操作符，方法是在导入列表中指定[注2]：

```
use autodie qw( open system :socket );
```

当autodie抛出错误时，它会把一个autodie::exception对象放到$@变量中，我们稍后就可以通过这个对象提取捕获的错误类型。下面的例子取自Paul Fenwick的autodie文档，使用given-when结构判别错误类型：

```
use 5.010;

open my $fh, '>', $filename; # 仍旧会在错误发生时调用 die 函数

given ($@) {
  when (undef)   { say "No error";                    }
  when ('open')  { say "Error from open";             }
  when (':io')   { say "Non-open, IO error.";         }
  when (':all')  { say "All other autodie errors."    }
  default        { say "Not an autodie error at all." }
  }
```

你也可以让autodie和Try::Tiny一起协作：

```
use 5.010;

use autodie;
use Try::Tiny;

try {
  open my $fh, '>', $filename; # 仍旧会在错误发生时启用 die 命令
  }
catch {
  when( 'open' ) { say 'Got an open error' }
  };
```

用grep筛选列表

有时候你可能希望选出列表中的部分成员，比如选出奇数，或者筛选文件中提到了Fred的行。在这一节你会看到，其实列表中的那些元素只需要用grep操作符即可筛选出来。

让我们先试试解决第一个问题，从一大堆数字中挑出奇数。不需引入任何新技术，我们可以这样写：

```
my @odd_numbers;
```

注2：　有关模块导入列表的详细说明，请参阅《Intermediate Perl》一书。

```
foreach (1..1000) {
  push @odd_numbers, $_ if $_ % 2;
}
```

这段代码用了取模操作符（%），我们之前在第二章介绍过这个操作符。如果某个数是偶数，那么这个数除以2的余数就是0，也就是假。而奇数会得到1，也就是真，因此只有奇数会被push到数组@odd_numbers中。

其实这段代码并没什么问题，只是不够简练高效。既然Perl提供了grep操作符，何不用它实现过滤器的功能：

```
my @odd_numbers = grep { $_ % 2 } 1..1000;
```

这么简短的代码得到的是一个包含500个奇数的列表。它是如何做到的呢？grep的第一个参数是代码块，使用$_作为列表的每个元素的占位符并返回真或假值。而代码块之后的参数则是将要被筛选的元素列表。grep操作符对列表的每个元素算出代码块的值，好像之前版本的foreach循环做的那样。代码块计算结果为逻辑真的那些元素，将会出现在grep返回的列表中。

在grep运行过程中，$_会轮流成为列表中每个元素的别名。这和之前看到的foreach循环是一样的。因此如果在grep表达式中修改$_的内容通常不是好主意，因为会破坏原始数据。

grep操作符和经典Unix工具同名，这个工具会使用正则表达式来从文件中找出匹配成功的行。这个任务也能用Perl的grep完成，并且威力更强大些。现在我们从一个文件中取出包含fred的行：

```
my @matching_lines = grep { /\bfred\b/i } <$fh>;
```

grep还有一个更为简单的写法。如果选择器(selector)需要的只是一个简单的表达式（而不是整个代码块），那么只要在这个表达式后面用逗号结束就可以了。下面就是刚才例子的简化版本：

```
my @matching_lines = grep /\bfred\b/i, <$fh>;
```

grep操作符在标量上下文中返回的是符合过滤条件的元素个数。在只需要统计文件中符合特定条件的行的数量而不必关心每行内容的时候，就可以采取这种用法。原本我们会先提取符合条件的行存到@matching_lines数组，再行统计：

```
my @matching_lines = grep /\bfred\b/i, <$fh>;
my $line_count = @matching_lines;
```

现在完全可以跳过保存中间数组的过程（也就不必创建数组并分配内存），直接通过标量赋值操作实现：

```
my $line_count = grep /\bfred\b/i, <$fh>;
```

用map把列表元素变形

除了作为过滤器之外，对于列表还有一项经常要做的工作，那就是把列表数据变形。举个例子，假设有一堆数字需要格式化成“金额数字”输出，就像第十三章里面&big_money子程序的做法那样。你不应该修改原始数据，你需要的是用于输出的修改后的列表拷贝。下面是传统做法：

```
my @data = (4.75, 1.5, 2, 1234, 6.9456, 12345678.9, 29.95);
my @formatted_data;

foreach (@data) {
  push @formatted_data, &big_money($_);
}
```

这段代码和grep那节开头的代码看起来很像，不是么？所以，如果把它改写成类似grep版本的应该不会太令人惊奇：

```
my @data = (4.75, 1.5, 2, 1234, 6.9456, 12345678.9, 29.95);

my @formatted_data = map { &big_money($_) } @data;
```

map操作符和grep非常相似，因为它们有同样的参数：一个使用$_的代码块和一个待处理的列表。而且它们的工作方式也非常相似，都会将$_设成列表每个元素的别名并且逐个为它们执行代码块。但map使用代码块中最后一个表达式的方式和grep不同，它返回的不是逻辑真假值，而是该表达式的实际计算结果，最终返回一系列这样的结果组成的列表。另一个重要差别是，map使用的表达式是在列表上下文中求值的，所以每次可以返回一个以上的元素。

任何形式的grep或map语句都可以改写成foreach循环，但中间需要借助临时数组保存数据。而实际上，简短版本更为高效，写起来也更方便。由于map或grep返回的结果是列表，所以可以直接把结果传递给其他函数作为参数。下面的例子就是在标题行之后，每行打印一个“金额数字”的做法：

```
print "The money numbers are:\n",
  map { sprintf("%25s\n", $_) } @formatted_data;
```

当然，实际上不用临时数组@formatted_data也行，直接从原始数据计算：

```
my @data = (4.75, 1.5, 2, 1234, 6.9456, 12345678.9, 29.95);
```

```
print "The money numbers are:\n",
  map { sprintf("%25s\n", &big_money($_) ) } @data;
```

和我们在grep中看到的一样，我们也可以用更简单的语法实现单条语句的map操作。如果你的选择器只需要一个简单的表达式（而不是一整个语句块）就能完成的话，可以直接写上这个表达式，跟上逗号即可：

```
print "Some powers of two are:\n",
  map "\t" . ( 2 ** $_ ) . "\n", 0..15;
```

更花哨的列表工具

在Perl里面还有一些模块专门用于列表数据的处理。毕竟，许多程序说到底不过是一系列对列表数据的变形罢了。

List::Util模块包含在标准库中，它能提供各式高效的常见列表处理工具，都是用C语言实现的。

比如我们想要知道某个列表中是否有元素符合特定条件。我们不必取出所有元素，只要找到第一个符合条件的就可以结束遍历，返回结果。所以这里我们不能用grep实现，因为它会对列表进行完整扫描，要是原始列表相当长的话，grep可能会多做许多无谓的工作：

```
my $first_match;
foreach (@characters) {
  if (/\bPebbles\b/i) {
    $first_match = $_;
    last;
  }
}
```

这样的代码实在太啰嗦，我们可以用List::Util提供的first子程序完成相同任务：

```
use List::Util qw(first);
my $first_match = first { /\bPebbles\b/i } @characters;
```

在第四章的习题中，你创建了一个名为&total的子程序，如果早知道List::Util的话，就不用费这么多事了：

```
use List::Util qw(sum);
my $total = sum( 1..1000 ); # 得到总和 500500
```

另外在第四章中的&max子程序也是如此，从列表中选出值最大的那个。这样的功能其实完全不必自己来写，直接用List::Util提供的版本就好了：

```
use List::Util qw(max);
my $max = max( 3, 5, 10, 4, 6 );
```

这个max只能处理数字，如果要处理字符串（使用字符串比较）的话，可以用maxstr实现：

```
use List::Util qw(maxstr);
my $max = maxstr( @strings );
```

如果要对列表中的元素随机排序的话，可以用shuffle实现：

```
use List::Util qw(shuffle);
my @shuffled = shuffle(1..1000); # 使列表元素的排序随机
```

另外还有一个模块，List::MoreUtils，它提供的工具更多。但它不是Perl自带的，所以你得自己到CPAN下载安装。你可以用它检查列表中是否没有一个元素，或者有任何元素，或者所有元素都符合特定条件。每个这样的列表工具都支持类似grep那样的语法：

```
use List::MoreUtils qw(none any all);

if (none { $_ > 100 } @numbers) {
  print "No elements over 100\n"
} elsif (any { $_ > 50 } @numbers) {
  print "Some elements over 50\n";
} elsif (all { $_ < 10 } @numbers) {
  print "All elements are less than 10\n";
}
```

如果需要对按元素组处理列表的话，可以用natatime（*N* at a time，同时处理*N*组）来取出对应位置上的元素：

```
use List::MoreUtils qw(natatime);

my $iterator = natatime 3, @array;
while( my @triad = $iterator->() ) {
  print "Got @triad\n";
}
```

如果要合并两个或多个列表，可以用mesh构造一个大型列表，交错填充原始列表中各个位置上的元素，就算其中某个列表长度很小都没关系：

```
use List::MoreUtils qw(mesh);

my @abc = 'a' .. 'z';
my @numbers = 1 .. 20;
my @dinosaurs = qw( dino );

my @large_array = mesh @abc, @numbers, @dinosaurs;
```

这会先取出@abc中的第一个元素并让它成为@large_array中的第一个元素，再取出@numbers中的第一个元素并让它成为@large_array中接下来的元素，然后按照同样的方式引入@dinosaurs中的数据。接着回到第一个@abc取出它的下一个元素，依此类推，直到所有列表中的元素全部取出为止。所以最终放在@large_array中的列表的开头的几个元素是：

```
a 1 dino b 2 c 3 ...
```

List::MoreUtils里面还有许多有意思的列表处理子程序，在自己实现具体程序之前，不妨先看看它的文档，如果有现成的，直接拿来用。

习题

下列习题答案参见第340页上的“第十七章习题解答”一节：

1. [30]写一个程序，从文件中读取一组字符串（每行一个），然后让用户键入模式以便进行字符串匹配。对每一个模式，程序应该说明文件里共有多少字符串匹配成功，分别是哪些字符串。对于所键入的每个新模式，不应重新读取文件，应该把这些字符串存放在内存里。文件名可以直接写在程序里。假如某个模式不合法（例如：括号不对称），那么程序应该汇报这些错误，并且让用户继续尝试其他模式。假如用户键入的不是模式而是空白行，那么程序就该停止运行。如果你需要一个充满有趣字符串的文件来进行匹配，那么试试*sample_text*这个文件吧。你应该已经从O'Reilly的网站下载过这个文件了。下载方式请在本书前言中查找。

2. [15]写一个程序，报告当前目录下所有文件的最后访问时间和最后修改时间（纪元时间），单位为秒。用stat取得文件的时间戳信息，利用列表切片的写法提取这两个元素。然后按照下面的三列形式输出结果：

```
fred.txt        1294145029      1290880566
barney.txt      1294197219      1290810036
betty.txt       1287707076      1274433310
```

3. [15]修改上题程序，把时间格式改为YYYY-MM-DD的形式。用map逐个输出，并用localtime通过列表切片提取纪元时间的年、月、日字段。注意localtime文档中对它返回的年份和月份数字的说明。最终输出的结果应该和下面类似：

```
fred.txt        2011-10-15      2011-09-28
barney.txt      2011-10-13      2011-08-11
betty.txt       2011-10-15      2010-07-24
```

附录 A 习题解答

本附录包含前面各章习题的答案。

第一章习题解答

1. 这个练习很简单，程序我们已经给你了。

   ```
   print "Hello, world!\n";
   ```

   如果你用的是Perl 5.10或以上版本，可以试试say：

   ```
   use 5.010;
   say "Hello, world!";
   ```

   如果希望在命令行尝试而不特地创建程序文件，可以用-e开关指定要执行的程序：

   ```
   $ perl -e 'print "Hello, World\n"'
   ```

   另外还有一个开关，-l，启用后会自动在输出内容后添加换行符：

   ```
   $ perl -le 'print "Hello, World"'
   ```

2. *perldoc*命令是和*perl*命令一起安装的，所以直接键入该命令就应该可以执行。

3. 这个程序也很容易，只要完成上面这道题，答案基本上就在这儿了：

   ```
   @lines = `perldoc -u -f atan2`;
   foreach (@lines) {
       s/\w<([^>]+)>/\U$1/g;
       print;
   }
   ```

第二章习题解答

1. 下面是实现方法之一：

```
#!/usr/bin/perl -w
$pi = 3.141592654;
$circ = 2 * $pi * 12.5;
print "The circumference of a circle of radius 12.5 is $circ.\n";
```

正如你所看到的，此程序以常见的#!行开头，你的机器上的Perl安装路径可能会有所不同。另外，我们也启用了警告信息。

程序代码正文里的第一行会将$pi的值设成我们需要的π的值。这种使用常量[注1]的做法有许多好处：在3.141592654重复出现时，可以节省键入时间；避免在某处使用3.141592654，却在另一处使用3.14159所造成的意外错误；你只需要检查一行程序代码，就可以避免因为不小心键入3.141952654而让宇宙飞船飞到别的星球去；键入$pi要比键入π容易得多，特别是在你没有Unicode的时候。并且，这样做在修改π值的时候也易于程序维护[注2]。接下来，我们会计算出圆周长并将它保存到$circ中，然后以漂亮的信息将其输出。信息最后面是换行符，因为只要是合格的程序，每行的输出都该以换行符做结尾。如果没有换行符，则视shell的提示符而定，输出结果可能会变成这样：

```
The circumference of a circle of radius 12.5 is
78.53981635.bash-2.01$[]
```

行尾的方框代表闪烁的光标，也就是shell在信息结尾的提示符[注3]。既然圆周长不应该是78.53981635.bash-2.01$，那么这应该算是程序的bug。因此，请务必在每一行输出的结尾加上\n。

2. 下面是实现方法之一：

```
#!/usr/bin/perl -w
$pi = 3.141592654;
print "What is the radius? ";
chomp($radius = <STDIN>);
$circ = 2 * $pi * $radius;
print "The circumference of a circle of radius $radius is $circ.\n";
```

注1： 如果你喜欢用更正式的方式来使用常量，那么你可以使用constant编译指令。

注2： π值的最近一次改变发生在一个多世纪之前，印第安纳州议会通过了该决议。请参阅1897年印第安纳州议会的第246号文件(*http://www.cs.uwaterloo.ca/~alopez-o/math-faq/node45.html*)。

注3： 我们问了O'Reilly，请他们帮我们用会闪烁的油墨来打印这个输入光标，我们愿意额外付钱，但他们不愿帮我们做。

此程序和前一题的类似，不过这次我们提示用户键入半径长度，然后用$radius（半径）代替前一题里写死的12.5。事实上，如果在写第一题的程序时能够考虑得更周全，当时我们也应该使用$radius这个变量。要注意的是，我们对输入值使用了chomp，就算不这么做，上面的算式依然有效，因为"12.5\n"之类的字符串会自动转换成数字12.5。但当我们要输出信息时，它看起来就会像这样：

```
The circumference of a circle of radius 12.5
 is 78.53981635.
```

我们发现，即使之前已经将$radius当成数字使用，但换行符还是会留在里面。因为print语句中的$radius和is中间有空格，所以输出的第二行开头也有空格。这个例子告诉我们，除非有特殊原因，否则请一律对输入值进行chomp处理。

3. 下面是实现方法之一：

```
#!/usr/bin/perl -w
$pi = 3.141592654;
print "What is the radius? ";
chomp($radius = <STDIN>);
$circ = 2 * $pi * $radius;
if ($radius < 0) {
  $circ = 0;
}
print "The circumference of a circle of radius $radius is $circ.\n";
```

在这里我们进一步检查了有问题的半径值。即使所输入的半径值不合理，程序至少也不会返回负的圆周长。你也可以先将半径设成0，再计算它的圆周长。办法不止一种。事实上，“办法不止一种（There Is More Than One Way To Do It!）”是Perl的座右铭。每道习题的解答都以“下面是实现方法之一”开头，道理就在此。

4. 下面是实现方法之一：

```
print "Enter first number: ";
chomp($one = <STDIN>);
print "Enter second number: ";
chomp($two = <STDIN>);
$result = $one * $two;
print "The result is $result.\n";
```

要注意的是，这个解答里省略了#!那行。事实上，接下来我们一律假设你已经知道它的存在，所以不用每次都重复提它。

上面的变量名称也许取得不太好。在长一点的程序里，程序维护员可能会认为$two的值应该是2。在这么短的程序里没什么关系，但如果程序很长，就该取比较有描述性的名称，像$first_response之类的。

在这个程序里，无论我们有没有对$one和$two这两个变量进行chomp都无关紧要，因为它们在赋值之后不会被当成字符串来用。可是，如果程序维护员在下星期修改

程序，让它输出像The result of multiplying $one by $two is $result.\n这样的信息，那么讨厌的换行符又会回来作祟。再次强调，除非有特殊原因（像下一题的情况），否则请一律对输入值进行chomp处理[注4]。

5. 下面是实现方法之一：

```
print "Enter a string: ";
$str = <STDIN>;
print "Enter a number of times: ";
chomp($num = <STDIN>);
$result = $str x $num;
print "The result is:\n$result";
```

从某个角度来看，这个程序和前一题的几乎完全相同。在这里，我们也是计算字符串的重复次数，因此保留了和前一题相同的程序结构。不过，这次我们不想对第一行输入字符串进行chomp，因为题目上要求将重复的每一行分开显示。这样一来，假设用户所输入的字符串是fred与换行符，而重复次数为3时，每行的fred后面就都会正确地加上换行符。

程序尾端的print语句里，我们将换行符放在$result的前面，这样第一行的fred才会以自成一行的方式被显示出来。换句话说，我们不想让所输出的三行fred中只有两行对齐，如下所示：

```
The result is: fred
fred
fred
```

这次我们不必在print输出的结尾加上换行符，因为$result应该已经以换行符结尾了。

程序里的空格在大部分情况下对Perl都没有影响，要不要加空格是你的自由。但请小心，别拼错字了！如果程序里的x和它前面的变量名称$str间没有空格，Perl所看到的将会是$strx，从而导致运行失败。

第三章习题解答

1. 下面是实现方法之一：

```
print "Enter some lines, then press Ctrl-D:\n"; # 或者试试 Ctrl-Z
@lines = <STDIN>;
@reverse_lines = reverse @lines;
print @reverse_lines;
```

注4： “chomp”这个词本身就是使劲哑巴着嘴嚼东西的意思，chomp处理就好比是咀嚼，虽然并非必须，但嚼一嚼再吞下去总没有坏处。

或者，更为简单的写法：

```
print "Enter some lines, then press Ctrl-D:\n";
print reverse <STDIN>;
```

除非所输入的列表在程序后面还会用到，否则大部分Perl程序员都会选择第二种做法。

2. 下面是实现方法之一：

```
@names = qw/ fred betty barney dino wilma pebbles bamm-bamm /;
print "Enter some numbers from 1 to 7, one per line, then press Ctrl-D:\n";
chomp(@numbers = <STDIN>);
foreach (@numbers) {
  print "$names[ $_ - 1 ]\n";
}
```

因为数组索引是从0数到6，所以这里必须将索引值减1，好让用户能够从1数到7。另外一种做法是在@names数组前面加上一个值来充数，像这样：

```
@names = qw/ dummy_item fred betty barney dino wilma pebbles bamm-bamm /;
```

如果你还额外检查了用户的输入是否在1到7的范围内，请给自己加分。

3. 如果想让所有的输出结果均显示在同一行，下面是实现方法之一：

```
chomp(@lines = <STDIN>);
@sorted = sort @lines;
print "@sorted\n";
```

或者，让每行分开显示：

```
print sort <STDIN>;
```

第四章习题解答

1. 下面是实现方法之一：

```
sub total {
  my $sum;  # 私有变量
  foreach (@_) {
    $sum += $_;
  }
  $sum;
}
```

这个子程序使用$sum来存储到目前为止的总和。每次子程序开始执行时，$sum都会是新创建的变量，因此其值为undef。之后，foreach循环会以$_作为控制变量来逐项处理@_里的参数列表。（请注意：我们再次强调参数数组@_跟foreach循环的默认变量$_之间并没有任何自动产生的联系。）

foreach循环第一次执行时，会将第一项参数（存储在$_中）与$sum相加。此时$sum的值还是undef，因为它还没有被存入任何东西。但是，由于Perl能从数字操作符+=判断它是被当成数字使用，所以会把它的值当作0，然后再将总和存回$sum里。

当循环再次执行时，下一项参数也会与$sum相加，而这时$sum的值已经不是undef了。两者的总和又会存回$sum里，然后再以相同的方式处理接下来的参数。全部处理完毕之后，程序的最后一行会将$sum返回给调用者。

对某些人来说，这个子程序可能有缺陷。假设子程序被调用时有一个空的参数列表，像在第四章正文中重写的子程序&max。这样，$sum的值将会是undef，也就是此子程序所返回的值。但在这个子程序里，比较恰当的做法是将空列表的总和设成0而非undef。（当然，如果你认为空列表的总和应该与(3,-5,2)的总和有所区别，那么返回undef是正确的做法。）

如果不想看到未定义的返回值，办法很简单：请直接将$sum的初始值设为0，而不是默认的undef：

```
my $sum = 0;
```

如此一来，即使参数列表是空的，这个子程序也一定会返回定义过的数字。

2. 下面是实现方法之一：

```
# 记得加上前一题里&total子程序的代码!
print "The numbers from 1 to 1000 add up to ", total(1..1000), ".\n";
```

要注意的是，我们不能在双引号括住的字符串里直接调用子程序[注5]，所以子程序调用是print的另一个独立参数。总和应该是500500，一个很好看的整数。程序的运行应该花不了多少时间，传递1 000个参数对Perl而言是常见的小事。

3. 下面是实现方法之一：

```
sub average {
  if (@_ == 0) { return }
  my $count = @_;
  my $sum = total(@_);               # &total来自前面的习题
  $sum/$count;
}

sub above_average {
  my $average = average(@_);
  my @list;
  foreach my $element (@_) {
    if ($element > $average) {
      push @list, $element;
```

注5： 也就是说，我们得用些高级技巧才可以做到。一般来说在Perl里没有什么是你完全无法做的。

```
        }
    }
    @list;
}
```

在average里，如果参数列表是空的，子程序就会结束，但并没有明确写上返回值。因此调用者将会取得undef [注6] 这个返回值，这表示空列表没有平均值。如果参数列表不是空的，那么&total就能帮忙计算平均值。此处并无必要使用$sum与$count这两个临时变量，但它们能让程序变得容易阅读些。

第二个子程序above_average会建立并返回由期望的元素构成的列表。（为何循环的控制变量是$element，而不是Perl最爱的默认变量$_？）请注意，这个子程序对于空参数列表有不同的处理方式。

4. 要记住greet上一次对话的人，可以使用一个state变量。一开始它会是undef，这样我们就知道Fred是它第一个问候的人。在这个子程序的结尾，我们把当前的$name保存在$last_name中，这样下一次我们才能记得它是什么：

```
use 5.010;

greet( 'Fred' );
greet( 'Barney' );

sub greet {
    state $last_person;

    my $name = shift;

    print "Hi $name! ";

    if( defined $last_person ) {
        print "$last_person is also here!\n";
    }
    else {
        print "You are the first one here!\n";
    }

    $last_person = $name;
}
```

5. 下面这种方法和前面的差不多，但是这次我们把所有出现过的名字都保存下来。我们不使用标量变量，而是用@names这个状态变量来保存所有的名字：

```
use 5.010;

greet( 'Fred' );
greet( 'Barney' );
greet( 'Wilma' );
```

注6： 如果&average在列表上下文中的话会返回一个空列表。

```
greet( 'Betty' );

sub greet {
    state @names;

    my $name = shift;

    print "Hi $name! ";

    if( @names ) {
        print "I've seen: @names\n";
    }
    else {
        print "You are the first one here!\n";

    }

    push @names, $name;
}
```

第五章习题解答

1. 下面是实现方法之一：

```
print reverse <>;
```

嗯，蛮简单的！能够这样写，是因为print的参数是所要输出的字符串列表，也就是在列表上下文中调用reverse的结果。reverse的参数是要被倒置的字符串列表，也就是在列表上下文中调用钻石操作符（diamond operator）的结果。钻石操作符所返回的列表是由用户选择的所有文件里的每一行所组成的。这个列表与*cat*命令所输出的结果相同。于是reverse会将此列表倒置，再交由print输出。

2. 下面是实现方法之一：

```
print "Enter some lines, then press Ctrl-D:\n";  # 或是 Ctrl-Z
chomp(my @lines = <STDIN>);

print "1234567890" x 7, "12345\n";  # 标尺行，到第 75 个字符的地方

foreach (@lines) {
  printf "%20s\n", $_;
}
```

此处，我们会先读取所有的文本行，再对它们进行champ处理。接下来，我们会输出标尺行(ruler line)。由于它是帮忙调试的工具，所以在程序写完之后我们通常会把它变成注释。我们可以重复键入"1234567890"，甚至使用复制与粘贴来制造出各种长度的标尺行，但我们还是选择了上面这种做法，因为比较酷。

接下来，foreach循环会逐项处理列表里的每行文本，将它们交由%20s转换后输出。还有另一种做法可以一次输出全部列表，而不必使用循环：

```
my $format = "%20s\n" x @lines;
printf $format, @lines;
```

这里有个常见的错误会让每行输出只有19个字符。假设你对自己说[注7]："嘿，既然最后会将换行符加回去，一开始又何必对输入做chomp呢？"于是就省略chomp，而把格式改成"%20s"（不含换行符）[注8]。然后程序运行时，奇怪的事发生了：输出结果少了一个空格。问题到底出在哪里？

这个做法会在Perl计算"需要多少个空格才有办法补齐所需字段的时候"发生问题。假设用户键入的是hello和换行符，则Perl所看到的是6个字符而不是5个，因为换行符也算一个字符。所以，Perl会输出14个空格以及含有6个字符的字符串，凑起来刚好就是你在"%20s"里所需要的20个字符。糟糕。

Perl判断字符串长度时当然不会去看它的内容，Perl只会检查字符数量。多余的换行符（或是其他特殊符号，像制表符或空字符）会造成意料之外的计算结果[注9]。

3. 下面是实现方法之一：

```
print "What column width would you like? ";
chomp(my $width = <STDIN>);

print "Enter some lines, then press Ctrl-D:\n";  # 或者 Ctrl-Z
chomp(my @lines = <STDIN>);

print "1234567890" x (($width+9)/10), "\n";      # 长度按需变化的标尺行

foreach (@lines) {
  printf "%${width}s\n", $_;
}
```

这个程序和前一题的相似，只不过这次会先问字段宽度。在程序一开始就询问，是因为在键入文件结尾指示符（end-of-file indicator）之后就不能再取得输入了（至少在某些系统上是如此）。当然，在实际读取用户输入时通常会用更好的输入结尾指示符（end-of-input indicator）。在后面的习题解答中会看到示例。

与前一题的另一个差异是在标尺行的处理上。按照附加题里的条件，我们使用了一些数学技巧，让标尺行至少和需要的长度相同。一个额外的挑战：你能证明这里的算式是正确的吗？（提示：考虑50和51两种宽度，然后别忘了x对右边的操作数是取整数，而不是四舍五入。）

注7：　或对Larry说，假如他站在你旁边的话。

注8：　除非Larry本人告诉过你别这么做。

注9：　现在Larry应该已经对你解释过了。

我们会用表达式"%${width}s\n"来产生这次的格式，其中使用$width进行内插。花括号是必要的隔离符号，可将变量名称与后面的s隔开；如果没有花括号，内插的就会是错误的变量$widths。如果你忘了怎么使用花括号，也可以使用'%'.$width."s\n"这样的表达式来产生相同的格式化字符串。

$width的值是另一个需要chomp的例子。如果没有对字段宽度进行chomp，最后的格式化字符串看起来就会像"%30\ns\n"，完全无效。

以前知道printf的人也许还会想到另一种解法。既然printf是从C语言借来的，而且C语言里并没有字符串变量内插，因此我们也可以使用C程序员的技巧。在转换字符串里，如果在应该放数字的地方出现星号（*），则可以使用参数列表里的值来替代，如下所示：

```
printf "%*s\n", $width, $_;
```

第六章习题解答

1. 下面是实现方法之一：

```
my %last_name = qw{
  fred flintstone
  barney rubble
  wilma flintstone
};
print "Please enter a first name: ";
chomp(my $name = <STDIN>);
print "That's $name $last_name{$name}.\n";
```

在这个程序里，我们使用qw//列表（以花括号为定界符）来初始化哈希。对于这个简单的数据设定而言并没有什么问题，因为数据的值是简单的名字与姓氏的配对，因此也很容易维护。但是，如果你的数据里含有空格，例如，如果robert de niro（罗伯特·德·尼罗）或mary kay place（玛丽·凯·普莱斯）访问Bedrock的话，这种简单的方法就不一定管用了。

你也可以将每一个键-值对分开设定，如下所示：

```
my %last_name;
$last_name{"fred"} = "flintstone";
$last_name{"barney"} = "rubble";
$last_name{"wilma"} = "flintstone";
```

请注意，如果你打算使用my来声明哈希（可能因为采用了use strict），则必须在声明之后才可对元素进行赋值。你不能只对变量里的某部分使用my，如下所示：

```
my $last_name{"fred"} = "flintstone";  # 糟糕!
```

换句话说，my操作符只能声明独立的变量，不能用来声明数组或哈希里的元素。另

外，请注意词法变量$name是在chomp函数调用的括号内声明的。像这种需要时再声明my变量的做法在Perl程序里十分常见。

在这个程序里，chomp也是不可或缺的。如果有人键入了"fred\n"这5个字符，而我们又没有对它进行chomp，程序就会去找键值为"fred\n"的哈希元素，但却找不到。当然，chomp并不是万能的，如果所键入的是"fred \n"（后面多了一个空格），我们就没办法以目前为止所学到的技巧来判断用户想要的其实是fred。

如果你还检查了哈希的键是否存在（使用exists函数），以便在用户打错字时显示说明信息，请给自己加分。

2. 下面是实现方法之一：

```
my(@words, %count, $word);     # 声明变量（可以省略）
chomp(@words = <STDIN>);

foreach $word (@words) {
  $count{$word} += 1;          # 或是$count{$word} = $count{$word} + 1;
}

foreach $word (keys %count) {  # 或是 sort keys %count
  print "$word was seen $count{$word} times.\n";
}
```

在这里，我们一开始就声明了所有变量。这对用过Pascal等语言的人来说，可能会比“需要时再声明”要熟悉得多（在Pascal里，变量一定要在最前面声明）。当然，我们是假设use strict正在起作用所以才声明这些变量的。Perl在默认情形下并不需要这种声明。

接下来，我们会在列表上下文中使用行输入操作符<STDIN>将所有输入行读进@words里，然后一次对全部输入行进行chomp。这样一来，@words的内容就会是由所有输入的单词所组成的列表了（假设一切顺利，则每行只有一个单词）。

现在，第一个foreach循环会逐项处理各个单词。该循环中包含了整个程序里最重要的一行，它会将$count{$word}的值加上1，然后再存回$count{$word}。你也可以不使用+=操作符，而改用比较长的写法。不过，较短的写法会稍微有效率一点，因为Perl只需要在哈希里查询一次$word就行了[注10]。在第一个foreach循环里，每次出现的单词都会让$count{$word}的值加1。假设第一个单词是fred，那么$count{"fred"}的值就会加1。既然这是第一次用到$count{"fred"}，它的值自然是undef。不过，因为我们将它当成数字来用（利用数字操作符+=或是较

注10： 在Perl的有些版本中，这种较短写法还可以避免输出一个使用未定义值的警告信息，而较长写法将会输出此警告信息。虽然我们还没提到，你也可以对变量使用++操作符来避免这种警告信息。

长写法的+），所以Perl会自动把undef转换为0。相加的总和为1，所以会将1存回$count{"fred"}。

在下一次foreach循环执行时，假设这次的单词是barney。我们会将$count{"barney"}的值加1，让它也从undef变成1。

现在，假设下一次的单词又是fred。我们会将$count{"fred"}的值（也就是1）再加上1而得到2。$count{"fred"}的值于是成为2，表示到目前为止fred出现过两次。

处理完第一个foreach循环之后，我们已经计算出了每个单词的出现次数。哈希键就是来自输入的单词，而相应的哈希值则是单词的出现次数。

最后，第二个foreach循环会逐项处理各个哈希键，也就是所有互不重复的单词。在这个循环里，每个不同的单词各会出现一次，而且每次会输出像“fred was seen 3 times”这样的信息。

附加题的解答：你可以在keys前面加上sort以便按照顺序输出哈希键。在输出结果超过十几行时将结果排序通常是件好事，它可以让调试的人迅速找到想要的条目。

3. 下面是实现方法之一：

```
my $longest = 0;
foreach my $key ( keys %ENV ) {
    my $key_length = length( $key );
    $longest = $key_length if $key_length > $longest;
    }

foreach my $key ( sort keys %ENV ) {
    printf "%-${longest}s  %s\n", $key, $ENV{$key};
    }
```

在第一个foreach循环中，我们会遍历所有哈希键并使用length函数得到它们的长度。如果当前的哈希键长度比保存在$longest变量中的长度还长，那么我们就把更长的那个值保存在变量$longest中。

一旦我们遍历完所有的哈希键，我们就可以用printf函数把键和值分两列打印出来。这里使用在第五章的第三个习题中用过的技巧，即使用变量内插的方式将$longest替换进模板字符串中。

第七章习题解答

1. 下面是实现方法之一：

```
while (<>) {
  if (/fred/) {
    print;
```

```
    }
  }
```

十分简单。本习题的重点其实是让你亲手试试例子里的各个字符串。它不会匹配Fred，这表示正则表达式会区分大小写（我们稍后会提到如何不区分大小写）。它会匹配frederick和Alfred，因为这两个字符串里都含有fred这4个字符（我们稍后会提到，如何比对独立的单词，让它不匹配frederick和Alfred）。

2. 下面是实现方法之一：把第一题的答案里的模式改成/[fF]red/。除此之外，也可以试试/(f|F)red/或/fred|Fred/，不过使用字符集的话性能更好。

3. 下面是实现方法之一：把第一题的答案里的模式改成/\./。必须加上反斜线（因为点号是元字符），或是写成字符集/[.]/。

4. 下面是实现方法之一：把第一题的答案里的模式改成/[A-Z][a-z]+/。

5. 下面是实现方法之一：把第一题的答案里的模式改成/(\S)\1/。\S字符集会匹配所有的非空白符，而圆括号可以让你使用反向引用\1来匹配紧跟着它的同样的字符。

6. 下面是实现方法之一：

```
while (<>) {
  if (/wilma/) {
    if (/fred/) {
      print;
    }
  }
}
```

此程序只有在/wilma/匹配成功时才会测试/fred/是否匹配。不过，fred可以在wilma之前出现，也可以在它之后出现。这两项测试是互相独立的。

如果你想要省略掉第二层的if测试，也可以像下面这么写[注11]：

```
while (<>) {
  if (/wilma.*fred|fred.*wilma/) {
    print;
  }
}
```

之所以能这样写，是因为若不是wilma在fred之前出现，就是fred在wilma之前出现。如果我们只写了/wilma.*fred/，那么即使fred and wilma flintstone这一行中提到过wilma和fred，还是不会与此模式相匹配。

我们把这一题当作附加题，是因为许多人在这里有理解上的障碍。我们提到了正则

注11：知道逻辑与操作符（我们将在第十章中看到）的人可以在if条件表达式中同时对/fred/和/wilma/进行匹配测试。这样会更有效率，更容易扩展，比之前给出的方法好多了。但我们还没学过逻辑与操作符。

表达式里的“or”运算（也就是竖线符号“|”），但却从来没有提到过“and”运算。这是因为正则表达式里并没有“and”运算[注12]。如果你想知道两个表达式是否都匹配成功，那就对它们都进行一下测试。

第八章习题解答

1. 有一种很简单的做法，我们已经直接写在正文中了。其输出应为before<match>after，如果你的输出不是这样，那就说明你绕远路了。

2. 下面是实现方法之一：

   ```
   /a\b/
   ```

   （当然，这就是要用在模式测试程序里面的模式！）如果你的模式不幸匹配了barney，说明你可能需要使用单词边界锚位(word-boundary anchor)。

3. 下面是实现方法之一：

   ```
   #!/usr/bin/perl
   while (<STDIN>) {
     chomp;
     if (/(\b\w*a\b)/) {
       print "Matched: |$`<$&>$'|\n";
       print "\$1 contains '$1'\n";      # 多输出一行
     } else {
       print "No match: |$_|\n";
     }
   }
   ```

   这是稍微修改过的模式测试程序，除了模式不同外，也额外加了一行打印$1的程序代码。

   此处的模式在括号内使用了一对\b单词边界锚位[注13]，但即使写在括号外面，也完全没有任何差别。那是因为锚位只会对应到字符串中的某个位置，而不会对应到某个字符：它的宽度为零（不会被括号捕获到）。

4. 下面这道习题的解答和上面的差不多，但是用了不同的正则表达式：

   ```
   #!/usr/bin/perl
   ```

注12：但实际上还是有一些需要技巧和高级的方式来实现类似的“and”操作。通常这些方式会比Perl的逻辑与的效率要低，但也不一定，这得看Perl和它的正则表达式引擎会怎么优化。

注13：老实说，其实并没有必要写上第一个锚位，详细的原因与星号的贪婪特性有关，但此处不谈。没有第一个锚位可能会比较有效率，有第一个锚位则比较清晰，所以最后我们还是选择了清晰。

```
use 5.010;

while (<STDIN>) {
  chomp;
  if (/(?<word>\b\w*a\b)/) {
    print "Matched: |$`<$&>$'|\n";
    print "'word' contains '$+{word}'\n";      # 新的输出行
  } else {
    print "No match: |$_|\n";
  }
}
```

5. 下面是实现方法之一：

```
m!
  (\b\w*a\b)        # $1：某个以字母a结尾的英文单词
  (.{0,5})          # $2：后面接上的字符不超过5个
!xs                 # /x和/s修饰符
```

（因为现在使用了两个内存变量，所以别忘了补上显示$2的程序代码。如果你自己又将模式修改成只使用一个内存变量，请把多余的那一整行标为注释。）如果你的模式不再成功匹配wilma，也许在模式中需要把“零个以上的字符”改成“一个以上的字符”。/s修饰符可以暂时忽略，因为数据里面应该没有换行符（当然，如果有的话，/s修饰符可能会产生不同的输出）。

6. 下面是实现方法之一：

```
while (<>) {
  chomp;
  if (/\s\z/) {
    print "$_#\n";
  }
}
```

井号（#）在这里用作标示字符，表示行尾的位置。

第九章习题解答

1. 下面是实现方法之一：

```
/($what){3}/
```

在$what替换完成后，会产生类似/(fred|barney){3}/的模式。如果省略圆括号，模式会变成/fred|barney{3}/，这也就等于是/fred|barneyyy/。因此，圆括号是不可或缺的。

2. 下面是实现方法之一：

```
my $in = $ARGV[0];
if (! defined $in) {
```

```
    die "Usage: $0 filename";
  }

  my $out = $in;
  $out =~ s/(\.\w+)?$/.out/;

  if (! open $in_fh, '<', $in ) {
    die "Can't open '$in': $!";
  }

  if (! open $out_fh, '>', $out ) {
    die "Can't write '$out': $!";
  }

  while (<$in_fh>) {
    s/Fred/Larry/gi;
    print $out_fh $_;
  }
```

此程序一开始会先清点它的命令行参数，预期应该要有一个。如果没有取得，就抱怨一下；如果取得，则把参数复制到$out并把扩展名换成.out（其实直接把文件名附加上.out就行了）。

在IN和OUT这两个文件句柄都被打开之后，才是程序最主要的部分。如果你没有使用/g和/i这两个修饰符，请自行扣半分，因为这样一来就没办法换掉所有的fred和Fred了。

3. 下面是实现方法之一：

```
  while (<$in_fh>) {
    chomp;
    s/Fred/\n/gi;          # 将所有的 FRED 替换为临时的占位符
    s/Wilma/Fred/gi;       # 将所有的 WILMA 替换为 Fred
    s/\n/Wilma/g;          # 再将所有占位符换回为 Wilma
    print $out_fh "$_\n";
  }
```

请把上题程序的循环换成这段循环。要进行这种互换，我们必须先找到一个“占位符”，而且必须是不会出现在数据中的。因为使用了chomp（最后输出的时候会补上一个换行符），所以我们知道换行符（\n）是绝对不会出现在字符串中的，所以换行符就可以充当占位符。NUL字符（\0）也是另一个不错的选择。

4. 下面是实现方法之一：

```
  $^I = ".bak";          # 制作备份
  while (<>) {
    if (/\A#!/) {          # 是#!开头的那行吗?
      $_ .= "## Copyright (C) 20XX by Yours Truly\n";
    }
    print;
  }
```

运行此程序时，应该在命令行参数中指定需要更新的文件。假设你的习题文件名都以ex...开头，比如*ex01-1*、*ex01-2*，那么你可以这样执行命令：

```
./fix_my_copyright ex*
```

5. 为了避免重复加上版权声明，我们得分两回处理所有文件。第一回，我们会先建立一个哈希，它的键是文件名称，而它的值是什么并不重要。为了简单起见，此处将值设为1：

```
my %do_these;
foreach (@ARGV) {
  $do_these{$_} = 1;
}
```

第二回，我们会把这个哈希当成待办事项列表逐个处理，并把已经包含版权声明行的文件移除。目前正在读取的文件名称可用$ARGV取得，所以可以直接把它拿来当哈希键：

```
while (<>) {
  if (/\A## Copyright/) {
    delete $do_these{$ARGV};
  }
}
```

最后的部分就跟之前所写的程序一样，但我们会事先把@ARGV的内容改掉：

```
@ARGV = sort keys %do_these;
$^I = ".bak";          # 准备备份
while (<>) {
  if (/\A#!/) {        # 是#!开头的那行吗?
    $_ .= "## Copyright (c) 20XX by Yours Truly\n";
  }
  print;
}
```

第十章习题解答

1. 下面是实现方法之一：

```
my $secret = int(1 + rand 100);
# 在调试时，可以去掉下面这行注释
# print "Don't tell anyone, but the secret number is $secret.\n";

while (1) {
  print "Please enter a guess from 1 to 100: ";
  chomp(my $guess = <STDIN>);
  if ($guess =~ /quit|exit|\A\s*\z/i) {
    print "Sorry you gave up. The number was $secret.\n";
    last;
  } elsif ($guess < $secret) {
    print "Too small. Try again!\n";
```

```
    } elsif ($guess == $secret) {
      print "That was it!\n";
      last;
    } else {
      print "Too large. Try again!\n";
    }
  }
```

此程序第一行会从范围1到100挑出一个秘密数字，运作细节如下。首先，rand是Perl的随机数函数，所以rand 100会产生0以上100以下的随机数。也就是说，该表达式的最大值差不多是99.999 [注14] 。加1之后，数字的范围将会是1到100.999，然后使用int函数取出整数部分，这就是我们所需要的范围1到100的数字。

放在注释后面的程序代码可协助程序的开发与调试，也可以帮你作弊。程序的主要部分是无限的while循环。执行到last之前，它会让我们不断猜下去。

测试数字之前先测试字符串，这一点很重要。如果我们不这样做，你猜得出用户键入quit时会怎么样吗？它会被解释成数字（如果启用警告功能，就会显示警告信息），因为它作为数字使用时是0，可怜的用户会收到“数字太小”的信息。这样的话，我们可能根本执行不到字符串测试的部分。

这里的无限循环还有另外一种写法，就是使用裸块及redo。这么写既不会执行得比较慢，也不会比较快，只是写法不同而已。一般来说，如果大部分时候会继续循环，就应该使用while，因为它默认会继续循环。如果只有在例外状况下才会继续循环，那么裸块也许是比较好的选择。

2. 这个程序是在之前的解答基础上做了少量的修改。我们需要在程序开发过程中打印秘密数字，所以在$Debug变量为真的时候调用print。而$Debug的值要么来自于环境变量，要么是默认值1。通过使用//操作符，我们在$ENV{DEBUG}未定义的时候设置它为1：

```
use 5.010;

my $Debug = $ENV{DEBUG} // 1;

my $secret = int(1 + rand 100);

print "Don't tell anyone, but the secret number is $secret.\n"
    if $Debug;
```

如果不用Perl 5.10的新特性，就必须做些额外工作：

```
my $Debug = defined $ENV{DEBUG} ? $ENV{DEBUG} : 1;
```

注14： 真正的上限视你的系统而定。如果你真想知道，参见*http://www.cpan.org/doc/FMTEYEWTK/random*。

3. 下面是实现方法之一，参考了第六章中习题三的答案。

在程序开头，我们设置了环境变量的值。键ZERO和EMPTY有假值（而非未定义值），而键UNDEFINED没有值。

之后在printf的参数列表中，使用//操作符来只在$ENV{$key}为未定义值的时候打印字符串(undefined)：

```
use 5.010;

$ENV{ZERO}      = 0;
$ENV{EMPTY}     = '';
$ENV{UNDEFINED} = undef;

my $longest = 0;
foreach my $key ( keys %ENV )
    {
    my $key_length = length( $key );
    $longest = $key_length if $key_length > $longest;
    }

foreach my $key ( sort keys %ENV )
    {
    printf "%-${longest}s  %s\n", $key, $ENV{$key} // "(undefined)";
    }
```

通过使用//，可以确保对ZERO和EMPTY键对应的假值不做处理。

若不使用Perl 5.10的功能，也可以使用条件操作符：

```
printf "%-${longest}s  %s\n", $key,
        defined $ENV{$key} ? $ENV{$key} : "(undefined)";
```

第十一章习题解答

1. 这个答案里用了哈希引用（关于哈希引用请参阅《Intermediate Perl》一书），但我们在此提供了可以不用它的方法。只要你知道怎么用，暂时别担心运作细节。先把事情办完，再慢慢来学习和了解）。

下面是实现方法之一：

```
#!/usr/bin/perl

use Module::CoreList;

my %modules = %{ $Module::CoreList::version{5.006} };

print join "\n", keys %modules;
```

2. 从CPAN安装DateTime模块后，你只需按要求创建两个日期对象，然后两者相减。但要注意前后顺序不要弄错：

```
use DateTime;

my $t = localtime;

my $now = DateTime->new(
    year       => $t[5] + 1900,
    month      => $t[4] + 1,
    day        => $t[3],
    );

my $then = DateTime->new(
    year       => $ARGV[0],
    month      => $ARGV[1],
    day        => $ARGV[2],
    );

my $duration = $now - $then;

my @units = $duration->in_units( qw(years months days) );

printf "%d years, %d months, and %d days\n", @units;
```

如果选用Time::Piece模块，就不必对照字段位置选取字段了，这时候的localtime返回的是时间对象，可以按照字段名返回年份数字和月份数字：

```
use Time::Piece;

my $t = localtime;

my $now = DateTime->new(
    year       => $t->year,
    month      => $t->mon,
    day        => $t->mday,
    );
```

要做得再漂亮一点，可以检查一下你键入的日期是否已经过去（否则日期相减得到的间隔时间会是一个负数，虽然也不是什么大问题）。比较数字大小的操作符同样可用于比较日期大小：

```
if( $now < $then ) {
    die "You entered a date in the future!\n";
    }
```

第十二章习题解答

1. 下面是实现方法之一：

```
foreach my $file (@ARGV) {
  my $attribs = &attributes($file);
  print "'$file' $attribs.\n";
}
```

```
sub attributes {
  # 报告某个给定文件的属性
  my $file = shift @_;
  return "does not exist" unless -e $file;

  my @attrib;
  push @attrib, "readable" if -r $file;
  push @attrib, "writable" if -w $file;
  push @attrib, "executable" if -x $file;  return "exists" unless @attrib;
  'is ' . join " and ", @attrib;  # 返回值
}
```

在这个例子里，使用子程序仍然是比较方便的做法。对于每个文件，主循环会用一行来输出它的属性，也许它会告诉我们'cereal-killer' is executable或者'sasquatch' does not exist。

上面的子程序会告诉我们某个文件的属性。当然，如果文件根本不存在，那就不需要进行其他测试了。因此我们会先测试它是否存在，如果不存在，就提早返回。

如果文件确实存在，我们将会建立一个列表用来存储文件的属性（如果你用特殊的_文件句柄而非$file，以避免重复调用系统来测试每项属性的话，请给自己加分）。要（效仿上面三个测试）加入新的测试是很简单的。但是，如果所有测试都不成功呢？嗯，即使不能说什么别的，起码可以说它存在，所以我们就这样做了。如果@attrib里有任何元素的话，它的值就会是真（这是在布尔上下文里，一种特殊的标量上下文）。这里的unless子句正是利用了这个事实。

不过，要是取得了某些属性，我们就可以用"and"把它们连接起来（使用join）并且在前面加上"is"，以造出is readable and writable这样的语句。这样的处理并不完美，如果有三个属性的话，它会说该文件is readable and writable and executable。虽然句子里的and有点多，但还算可以接受。如果想要再加上更多其他属性的测试，而你又在意这种事情的话，也许该将它的输出改成像is readable, writable, exeutable, and nonempty这样。

要注意的是，如果你碰巧没有在命令行键入任何文件名，则程序将不会有任何输出。这很合理：如果你查询零个文件的信息，本来就该得到零行的结果。但是，这种做法可以跟下一题的程序作个比较。

2. 下面是实现方法之一：

```
die "No file names supplied!\n" unless @ARGV;
my $oldest_name = shift @ARGV;
my $oldest_age = -M $oldest_name;

foreach (@ARGV) {
  my $age = -M;
  ($oldest_name, $oldest_age) = ($_, $age)
    if $age > $oldest_age;
```

```
}

printf "The oldest file was %s, and it was %.1f days old.\n",
  $oldest_name, $oldest_age;
```

程序一开始会检查文件名，如果没有取得任何文件名，就会显示错误信息。这是因为程序的用途是找出最旧的文件，如果没有取得文件名，自然就不会有最旧的文件。

我们再一次用到了“高水线（high-water-mark）”算法。第一个文件当然是目前唯一见过的文件中最旧的。我们必须记下它的年龄，并存储在$oldest_age变量里。

对于每个文件，我们都会像上面那样利用-M文件测试来取得它们的年龄（不过，这里用的是$_的默认参数）。一般所谓的文件“年龄”，通常指的是上次修改的时间，虽然你也可以做不同的解释。如果目前的文件年龄大于$oldest_age，我们就会以列表赋值的方式同时更新文件名和年龄变量的值。尽管不一定要采取列表赋值的方式，但它确实是一次更新数个变量的好方法。

此程序中，我们将-M返回的年龄存进临时变量$age里。如果不使用临时变量，每次都直接使用-M，又会怎样呢？首先，除非使用特殊的_文件句柄，否则我们每次都得向操作系统询问文件的年龄，这可能会多花一些时间（你大概不会注意到，除非有成千上百个文件；就算真的有这么多文件，也可能不会有什么影响）。不过更重要的是我们应该考虑到如果有人在我们进行检查的同时更新文件的话，该怎么办？也就是说，在我们第一次使用-M取得某个文件的年龄时，发现它是目前为止所看到的最旧的。但是，在我们第二次使用-M之前，有人修改了那个文件，将它的时间戳设成了当前的时间。这样一来，实际存进$oldest_age里的却可能是系统上年龄最轻的文件。程序运行的结果就会是自该文件之后最旧的文件，而不是全部文件中最旧的。这种问题调试起来会非常困难！

程序结尾处我们以printf输出文件名和年龄，并将天数取到小数点后一位。假如你将年龄转换成天数、小时数及分钟数来显示，请给自己加分。

3. 下面是实现方法之一：

```
use 5.010;

say "Looking for my files that are readable and writable";

die "No files specified!\n" unless @ARGV;

foreach my $file ( @ARGV ) {
    say "$file is readable and writable" if -o -r -w $file;
    }
```

使用栈式文件测试操作符的前提是用Perl 5.10及以上版本，因此我们使用use语句

开头来确保版本正确。我们检查@ARGV数组，确保其中有数据供foreach处理，否则调用die。

我们使用三个文件测试操作符：-o用来检查我们是否拥有文件，-r用来检查文件是否可读，而-w用来检查文件是否可写。把它们堆叠在一起成为-o -r -w可以一并检查是否通过，也就是我们需要的效果。

如果要用Perl 5.10之前的版本完成以上功能，也只是稍微多些代码而已。需要用print加上换行符来模拟say，并且用短路操作符&&来组合文件测试：

```
print "Looking for my files that are readable and writable\n";

die "No files specified!\n" unless @ARGV;

foreach my $file ( @ARGV ) {
    print "$file is readable and writable\n"
        if( -w $file && -r _ && -o _ );
    }
```

第十三章习题解答

1. 下面是实现方法之一，使用glob：

```
print "Which directory? (Default is your home directory) ";
chomp(my $dir = <STDIN>);
if ($dir =~ /\A\s*\Z/) {         # 空白行
  chdir or die "Can't chdir to your home directory: $!";
} else {
  chdir $dir or die "Can't chdir to '$dir': $!";
}

my @files = <*>;
foreach (@files) {
  print "$_\n";
}
```

首先，我们会显示一个简单的提示，然后取得用户想要的目录，对它进行必要的chomp（如果没有chomp，则会尝试转到一个名称结尾有换行符的目录。这在Unix上是合法的，所以chdir函数不能自动帮你把换行符去掉）。

接下来，如果目录名称不是空的，我们会切换到该目录下，遇到错误就中断执行。如果名称是空的，就以用户主目录来代替。

最后，使用星号的glob操作会返回（新）工作目录中所有的文件名，并自动按字母顺序排序，然后逐一输出。

2. 下面是实现方法之一：

```
print "Which directory? (Default is your home directory) ";
```

```
chomp(my $dir = <STDIN>);
if ($dir =~ /\A\s*\Z/) {         # 空白行
  chdir or die "Can't chdir to your home directory:
$!";
} else {
  chdir $dir or die "Can't chdir to '$dir': $!";
}

my @files = <.* *>;        ## 现在加上了 .*
foreach (sort @files) {    ## 现在排序
  print "$_\n";
}
```

和前一题有两点差别：第一，这次的glob操作包含了“点号星号”，它会匹配所有以点号开头的文件名；第二，我们必须对所得到的列表进行排序，因为取出的列表中，以点号开头的文件名会和不以点号开头的文件名交错排列，看起来比较凌乱，排序之后会清楚很多。

3. 下面是实现方法之一：

```
print 'Which directory? (Default is your home directory) ';
chomp(my $dir = <STDIN>);
if ($dir =~ /\A\s*\Z/) {         # 空白行
  chdir or die "Can't chdir to your home directory:
$!";
} else {
  chdir $dir or die "Can't chdir to '$dir': $!";
}

opendir DOT, "." or die "Can't opendir dot: $!";
foreach (sort readdir DOT) {
  # next if /\A\./; ## 如果跳过文件名以点号开头的文件
  print "$_\n";
}
```

这个程序的结构同前两题，但是现在我们改用打开目录句柄的方式。变更工作目录之后，我们会打开当前目录，也就是DOT目录句柄。

为什么要用DOT呢？如果用户键入了像/etc这样的绝对目录名称，那么打开它并没有什么问题。但是，如果用户键入了像fred这样的相对目录名称呢？让我们来看看会发生什么事。首先，让我们chdir到fred目录，然后再用opendir来打开fred。可是，这样会打开新目录里的fred，而不是原来目录里的fred。只有.总是表示“当前目录”（起码在Unix和类似的系统上是这样）。

readdir函数会取得目录里所有的文件名，然后再由程序将它们排序输出。如果以这种方式来做第一题，那么我们就应该略过文件名以点号开头的文件。要这么做，只需把foreach循环里的注释去掉就行了。

你也许会怀疑：“为什么要先chdir呢？readdir类型的函数不一定要当前目录，它

其实可以作用在任何目录上。”最主要的动机是想让用户只按一个键，就可以转移到其主目录。但是，这个程序也可以作为“通用文件管理器”的雏形。也许接下来我们可以设计一个功能，让用户选择要备份目录里的哪些文件等。

4. 下面是实现方法之一：

```
unlink @ARGV;
```

或者，如果想在程序遇到问题时对用户提出警告，也可以这样写：

```
foreach (@ARGV) {
  unlink $_ or warn "Can't unlink '$_': $!, continuing...\n";
}
```

在这里，来自命令调用行的每个条目都会被单独地放入$_，然后成为unlink的参数。如果其中出现问题，警告信息能提供线索。

5. 下面是实现方法之一：

```
use File::Basename;
use File::Spec;

my($source, $dest) = @ARGV;

if (-d $dest) {
  my $basename = basename $source;
  $dest = File::Spec->catfile($dest, $basename);
}

rename $source, $dest
  or die "Can't rename '$source' to '$dest': $!\n";
```

程序里实际做事的只有最后一行，其他的部分是为了“把文件移动到目录中”而存在的。先在一开始声明所用到的模块，再为命令行参数取有意义的名称。如果$dest是目录，我们需要从$source名称中取出文件基名，并将它附加到$dest后面。最后，一旦$dest经过必要的处理，rename函数会执行实际改名的动作。

6. 下面是实现方法之一：

```
use File::Basename;
use File::Spec;

my($source, $dest) = @ARGV;

if (-d $dest) {
  my $basename = basename $source;
  $dest = File::Spec->catfile($dest, $basename);
}

link $source, $dest
  or die "Can't link '$source' to '$dest': $!\n";
```

正如题目中所提示的，此程序和前一题的十分相似，唯一的差别在于这次执行的是link而非rename。如果你的系统不支持硬链接，则最后的语句可以改成这样：

```
print "Would link '$source' to '$dest'.\n";
```

7. 下面是实现方法之一：

```
use File::Basename;
use File::Spec;

my $symlink = $ARGV[0] eq '-s';
shift @ARGV if $symlink;

my($source, $dest) = @ARGV;
if (-d $dest) {
  my $basename = basename $source;
  $dest = File::Spec->catfile($dest, $basename);
}

if ($symlink) {
  symlink $source, $dest
    or die "Can't make soft link from '$source' to '$dest': $!\n";
} else {
  link $source, $dest
    or die "Can't make hard link from '$source' to '$dest': $!\n";
}
```

开头几行程序代码（在两个use声明之后）会先检查第一个命令行参数，如果它是-s，就表示所要建立的是软链接，所以我们将此判断的真假值存储在$symlink变量中。如果检查到了-s，我们还得将它去掉，也就是下一行程序代码所做的事。之后的数行程序代码是从上一题的解答复制过来的。最后，依照$symlink的值是真是假，程序会选择建立硬链接或软链接。最后，我们还更改了die后面的信息，让它清楚显示出我们试图建立的是哪一种链接。

8. 下面是实现方法之一：

```
foreach ( glob( '.* *' ) ) {
  my $dest = readlink $_;
  print "$_ -> $dest\n" if defined $dest;
}
```

glob操作所返回的每个条目都会依次作为$_的值。如果该条目是软链接，那么就会由readlink返回一个已定义的值并且输出链接位置；如果不是，测试条件就会失败，从而使得程序略过该条目。

第十四章习题解答

1. 下面是实现方法之一：

```
my @numbers;
push @numbers, split while <>;
foreach (sort { $a <=> $b } @numbers) {
  printf "%20g\n", $_;
}
```

程序代码的第二行实在令人困惑，不是吗？嗯，这是故意的。虽然我们建议你编写清楚易懂的程序代码，但也有人以写出复杂难解的程序为乐[注15]，所以你最好能预先做好准备。总有一天，你也会需要维护这种难懂的程序代码。

因为那一行用到了while修饰符，所以与下面的循环等效：

```
while (<>) {
  push @numbers, split;
}
```

这样好多了，但也许还是有点不清楚（不过，这种写法我们可以接受。它还没有越过“一眼难以看懂”这条线）。while循环每次会读入一行（从用户所要求的输入来源，也就是钻石操作符），接着（在默认的状况下）split会以空白来分割该行，于是会产生一个单词列表，也就是数字列表，毕竟这里的输入只不过是一系列以空白分隔的数字而已。这样一来，无论输入怎么排列，while循环都会将其中所有的数字存进@numbers里。

接下来，foreach循环会逐行输出排过序的列表，使用%20g数字格式来让它们靠右对齐。如果你使用%20s，又会怎样呢？嗯，因为后者是字符串格式，所以它不会更改输出中的字符串。你是否注意到样本数据里同时包含了1.50和1.5，以及04和4呢？如果你将它们当成字符串输出，那么多余的零字符还会留在输出结果里；但是%20g是数字格式，所以相等数字的呈现方式也会相同。这两种格式都有可能是对的，具体使用哪个要根据情况决定。

2. 下面是实现方法之一：

```
# 别忘了将哈希%last_name放在这里,
# 你可以从习题说明里照抄或是下载范例文件
my @keys = sort {
  "\L$last_name{$a}" cmp "\L$last_name{$b}"  # 按姓氏排序
   or
  "\L$a" cmp "\L$b"                           # 按名字排序
} keys %last_name;

foreach (@keys) {
```

注15：应该说，我们不建议你在日常写程序的时候使用，不过把编写令人困惑的程序当作游戏来玩还是挺有趣的，而花一两个周末来搞懂别人写的迷津程序（obfuscated program）也会令你受益匪浅。如果想看看这些程序或找人帮忙解码，请在下次Perl Monger大会上问问。你也可以在Web上搜索JAPH，或是看看自己能否解开第十四章结尾处的迷津程序。

```
  print "$last_name{$_}, $_\n";              # 打印：Rubble,Bamm-Bamm
}
```

对于这个程序没什么好解释的。它会依题目的要求对哈希键进行排序，然后输出。我们之所以会先输姓再输名，纯粹只为了好玩而已，题目里并没有指定要用哪种显示方式。所以此题的答案就留给你自己去解释了。

3. 下面是实现方法之一：

```
print "Please enter a string: ";
chomp(my $string = <STDIN>);
print "Please enter a substring: ";
chomp(my $sub = <STDIN>);

my @places;

for (my $pos = -1; ; ) {                    # 三节式for循环的技巧性用法
  $pos = index($string, $sub, $pos + 1);  # 找出下个位置
  last if $pos == -1;
  push @places, $pos;
}

print "Locations of '$sub' in '$string' were: @places\n";
```

这个程序的开头十分简单。它要求用户键入字符串，然后声明一个数组来存储子串出现的位置。但是接下来的for循环似乎又是个“精巧至上”的程序代码。做这种事好玩可以，但绝不应该在实际应用的程序里出现。不过，这里展示的技巧可能以后用得到，所以让我们来看看它是如何运作的。

用my来声明的$pos变量是for循环作用域内的私有变量，初始值为-1。这里我们就不再卖关子了，直接告诉你它的功能是存储在较长的字符串里子字符串的出现位置。for循环的“测试”和“递增”部分都是空的，所以这是个无限循环（当然，我们终究会脱离循环的，这次是用last）。

循环主体的第一行语句会从位置$pos+1开始寻找子字符串的出现位置。也就是说，在循环第一次执行，$pos还是-1的时候，会从位置0（字符串开头）开始寻找。接着把子字符串的出现位置存入$pos。如果它是-1，就不必再执行for循环了，所以我们会用last来脱离循环。如果$pos不是-1，我们就会将位置存进@places，然后再进行下一次循环。这时，$pos+1会让程序在继续寻找子字符串时，从上次出现的位置的后面一格开始。如此一来，我们得到了想要的解答，一切都又恢复到了原来的平静。

如果你不想使用这种奇妙的for循环，也可以用下面的写法来得到相同的结果：

```
{
  my $pos = -1;
  while (1) {
    ... # 循环体和上面代码中的相同
  }
```

```
}
```

外层的裸块限制住了$pos的作用域。你不一定得这么做，但在尽可能小的有效范围内声明变量通常是比较好的做法。这么做能减少程序里同时“活着”的变量，让我们得以降低之后不小心将$pos这个名称用到别处的可能性。基于同样的道理，如果不将变量声明在较小的作用域里，通常就应该给它取较长的名称，以避免之后不小心重复用到。在这个程序里，$substring_position就是一个不错的名字。

另一方面，如果你想让程序代码成为迷津（你真坏！），同一个程序也可以写成如下所示的大怪物（我们真坏！）：

```
for (my $pos = -1; -1 !=
  ($pos = index
    +$string,
    +$sub,
    +$pos
    +1
  );
push @places, ((((+$pos))))) {
    'for ($pos != 1; # ;$pos++) {
      print "position $pos\n";#;';#' } pop @places;
}
```

这份更刁钻古怪的程序代码可以代替原本程序里奇妙的for循环。到了这里，你的知识应该已经足以解读出它的意义了。现在，你也可以写出自己的迷津程序，让朋友吃惊，使敌人困惑。请将这份力量用以为善，不要作恶。

对了，假设你在This is a test.里寻找t的话，结果会是什么呢？它出现在10和13这两个位置；它并不会出现在位置0，因为它会区分大小写。

第十五章习题解答

1. 下面是实现方法之一，重写第十章中的猜数程序。我们不必使用智能匹配，但可以使用given：

```
use 5.010;

my $Verbose = $ENV{VERBOSE} // 1;

my $secret = int(1 + rand 100);

print "Don't tell anyone, but the secret number is $secret.\n"
    if $Verbose;

LOOP: {

    print "Please enter a guess from 1 to 100: ";
    chomp(my $guess = <STDIN>);
```

```
    my $found_it = 0;

    given( $guess ) {
        when( ! /\A\d+\Z/ )  { say "Not a number!" }
        when( $_ > $secret ) { say "Too High!"    }
        when( $_ < $secret ) { say "Too low!"     }
        default              { say "Just right!"; $found_it++ }
        }

    last LOOP if $found_it;
    redo LOOP;

}
```

在第一个when中，我们先检查我们是否有数字。如果没有数字字符，或整个字符串为空，就不用继续后面的数字比较了。

注意我们没有把last放在default块中。原先我们就是这么做的，但Perl 5.10.0会因此发出警告，所以现在去掉了（或许以后的新版本就不会有这个问题）。

2. 下面是实现方法之一：

```
use 5.010;

for (1 .. 105) {
    my $what = '';
    given ($_) {
        when (not $_ % 3) { $what .= ' Fizz'; continue }
        when (not $_ % 5) { $what .= ' Buzz'; continue }
        when (not $_ % 7) { $what .= ' Sausage' }
    }
    say "$_$what";
}
```

3. 下面是实现方法之一：

```
use 5.010;

for( @ARGV )
    {
    say "Processing $_";

    when( ! -e ) { say "\tFile does not exist!" }
    when( -r _ ) { say "\tReadable!"; continue }
    when( -w _ ) { say "\tWritable!"; continue }
    when( -x _ ) { say "\tExecutable!"; continue }
    }
```

如果在for块中直接使用when，就不必再写given了。接下来的程序先判断文件是否存在（其实是反过来判断的）。如果执行到第一个when块，就会报告文件不存在并靠隐式break跳出，以避免执行之后的when测试。

在第二个when当中，我们测试文件可读与否，这次用的是-r文件测试操作符。另外

还用到了特殊的虚拟文件句柄_来使用来自上一个stat文件的缓存信息（也就是文件测试如何得到它们的信息）。如果不写_，其实程序也能工作，只是测试更繁琐些。when块的最后使用了continue来跳入下一个when测试。

4. 下面是使用given和智能匹配的实现方法之一：

```
use 5.010;

say "Checking the number <$ARGV[0]>";

given( $ARGV[0] ) {
    when( ! /\A\d+\Z/ ) { say "Not a number!" }

    my @divisors = divisors( $_ );

    my @empty;
    when( @divisors ~~ @empty ) { say "Number is prime" }

    default { say "$_ is divisible by @divisors" }
    }

sub divisors {
    my $number = shift;

    my @divisors = ();
    foreach my $divisor ( 2 .. $number/2 ) {
        push @divisors, $divisor unless $number % $divisor;
        }

    return @divisors;
    }
```

我们首先汇报正在处理的数字。这个习惯很好，可以告诉大家程序还在运行。我们用尖括号来区分$ARGV[0]和字符串的其余部分。

在given中，我们写了两个when块，用来组织一些其他语句。前面的when通过尝试正则表达式匹配来判断我们确实有个数字。如果那个模式匹配失败，我们就用相应块内的代码输出“Not a number!”。这个when块有隐式break功能，可以退出整个given结构。如果测试通过，就能调用divisors()。这个调用其实可以写在given之外，但那样就可能有风险。万一传入的不是数字，好比是“Fred”,怎么办？为避免Perl产生警告信息，我们还是把when当成一种预警机制。

一旦除法结束，我们就希望知道@divisors中是否有什么数据。这时候当然可以在标量上下文中使用数组获取元素的数量，但也可以使用智能匹配。我们早就知道在比较两个数组的时候，必须要有同样的元素和同样的顺序。这里我们创建一个空数组@empty，自然没有任何元素。如果拿它和@divisors比较，智能匹配就只会在没有因数的时候才能成功。如果成功的话，就可以执行相应的when块，这一块也是靠隐式break退出的。

最终通过测试的数一定不是质数，所以用default块来打印整除数字列表。

这里还有些更加精彩的内容，虽然我们承诺在本书中不提引用，而是留到《Intermediate Perl》再说。这里我们通过新建一个空数组来检查@divisors是否为空，但其实这是多余的，只要用匿名数组一次便可成形：

```
when( @divisors ~~ [] ) { ... }
```

5. 下面是基于上题的实现方法之一：

```
use 5.010;

say "Checking the number <$ARGV[0]>";

my $favorite = 42;

given( $ARGV[0] ) {
    when( ! /\A\d+\Z/ ) { say "Not a number!" }

    my @divisors = divisors( $ARGV[0] );

    when( @divisors ~~ 2 ) { # 如果 2 在 @divisors 里面

        say "$_ is even";
        continue;
        }

    when( !( @divisors ~~ 2 ) ) { # 如果 2 不在 @divisors 里面
        say "$_ is odd";
        continue;
        }

    when( @divisors ~~ $favorite ) {
        say "$_ is divisible by my favorite number";
        continue;
        }

    when( $favorite ) { # $_ ~~ $favorite
        say "$_ is my favorite number";
        continue;
        }

    my @empty;
    when( @divisors ~~ @empty ) { say "Number is prime" }

    default { say "$_ is divisible by @divisors" }
    }

sub divisors {
    my $number = shift;

    my @divisors = ();
    foreach my $divisor ( 2 .. ($ARGV[0]/2 + 1) ) {
        push @divisors, $divisor unless $number % $divisor;
```

```
    }

  return @divisors;
  }
```

这个扩展练习增加了更多的when块来完成更多的条件判断。一旦获得@divisors数组，就可以用智能匹配来检查其中内容。如果2在@divisors之中，就可以断定这是偶数。我们在报告之后用显式continue来驱动given结构执行之后的when测试。对于奇数的情况，同样使用智能匹配，只是对结果取反。同样的技巧还可以判断我们喜欢的数字是否在@divisors中，或者输入的恰巧就是我们喜欢的数字。

第十六章习题解答

1. 下面是实现方法之一：

```
chdir '/' or die "Can't chdir to root directory: $!";
exec 'ls', '-l' or die "Can't exec ls: $!";
```

第一行程序代码会将当前工作目录切换到根目录，它的名称总是固定的。第二行使用了多参数的exec函数来将结果传送到标准输出。我们也可以使用单参数的形式，但上面的做法并没什么不好。

2. 下面是实现方法之一：

```
open STDOUT, '>', 'ls.out' or die "Can't write to ls.out: $!";
open STDERR, '>', 'ls.err' or die "Can't write to ls.err: $!";
chdir '/' or die "Can't chdir to root directory: $!";
exec 'ls', '-l' or die "Can't exec ls: $!";
```

程序的前两行会重新打开STDOUT与STDERR并将它们重定向到当前工作目录下的两个文件里（在切换工作目录之前）。接下来，在工作目录切换之后，将会执行显示目录清单的命令，并把数据传送到之前打开的两个文件里。

最后一个die所显示的信息会出现在哪里呢？当然，它会跑到*ls.err*里，因为在那时STDERR已经被定向到该文件里了。至于chdir后面的die也会将信息传送到*ls.err*里。可是，如果在第二行上无法重新打开STDERR，错误信息又会在哪里出现呢？它会在原本的STDERR上出现。这是因为当STDIN、STDOUT、STDERR这三个标准文件句柄重新打开失败时，原先的文件句柄仍然会继续打开着。

3. 下面是实现方法之一：

```
if (`date` =~ /\AS/) {
  print "go play!\n";
} else {
  print "get to work!\n";
}
```

因为Saturday（周六）和Sunday（周日）都是以S开头，而且*date*命令的输出结果又是以“今天是星期几”作为开始，所以这个程序十分简单，只要检查*date*命令的输出，看看它是否以S开头就行了。还有许多更复杂的做法可以获得相同的结果，其中大部分我们都介绍过了。

不过，如果要实际应用这个程序，我们大概会将模式换成/\A(Sat|Sun)/。它会稍微慢一点点，但几乎没什么影响；再说，这对程序维护员而言要好懂多了。

4. 要捕获某些信号，就要设置信号处理句柄。和本书之前展示的技术那样，我们需要做一些重复性的工作。对每一个信号处理句柄子程序，我们都设置一个state变量，这样每次调用子程序时都可以访问之前累计的数字。我们用foreach循环依次将正确的子程序名称赋值给%SIG中的适当键。最后创建一个无限循环，让程序一直运行着，准备接受信号：

```
use 5.010;

sub my_hup_handler  { state $n; say 'Caught HUP: ',  ++$n }
sub my_usr1_handler { state $n; say 'Caught USR1: ', ++$n }
sub my_usr2_handler { state $n; say 'Caught USR2: ', ++$n }
sub my_usr2_handler { say 'Caught INT. Exiting.'; exit }

say "I am $$";

foreach my $signal ( qw(int hup usr1 usr2) ) {
    $SIG{ uc $signal } = "my_${signal}_handler";
    }

while(1) { sleep 1 };
```

我们需要另外打开一个终端会话来运行程序以发送信号：

```
$ kill -HUP 61203
$ perl -e 'kill HUP => 61203'
$ perl -e 'kill USR2 => 61203'
```

该程序的输出显示每次信号来到时我们已经见过的次数：

```
$ perl signal_catcher
I am 61203
Caught HUP: 1
Caught HUP: 2
Caught USR2: 1
Caught HUP: 3
Caught USR2: 2
Caught INT. Exiting.
```

第十七章习题解答

1. 下面是实现方法之一：

```
my $filename = 'path/to/sample_text';
open my $fh, '<', $filename
  or die "Can't open '$filename': $!";
chomp(my @strings = <FILE>);
while (1) {
  print 'Please enter a pattern: ';
  chomp(my $pattern = <STDIN>);
  last if $pattern =~ /\A\s*\Z/;
  my @matches = eval {
    grep /$pattern/, @strings;
  };
  if ($@) {
    print "Error: $@";
  } else {
    my $count = @matches;
    print "There were $count matching strings:\n",
      map "$_\n", @matches;
  }
  print "\n";
}
```

此程序使用eval块来捕获使用正则表达式时可能发生的错误。在eval块里，grep会筛选出字符串列表中匹配模式的字符串。

一旦eval执行完毕，程序就会汇报错误信息或显示匹配的字符串。请注意，为了在每个字符串后面加上换行符，我们使用map来对输出进行“unchomp”操作。

2. 这个程序非常简单。有许多种取得文件列表的方法，不过我们现在关心的只是当前工作目录内的文件，所以用glob提取就好了。我们通过foreach把取得的每个文件名存到默认的控制变量$_中，因为stat默认也是使用这个变量的，所以能省下很多事。在stat外围加上圆括号来取列表切片：

```
foreach ( glob( '*' ) ) {
  my( $atime, $mtime ) = (stat)[8,9];
  printf "%-20s %10d %10d\n", $_, $atime, $mtime;
  }
```

我们查阅stat的文档后知道，位于第8和第9索引位置上的就是我们需要的时间戳。文档作者已经非常贴心地标注了列表的每个索引位置上的字段意义，所以不用我们费力数就能知道该用哪个下标。

如果不想用$_，当然也可以自己指定控制变量：

```
foreach my $file ( glob( '*' ) ) {
  my( $atime, $mtime ) = (stat $file)[8,9];
  printf "%-20s %10d %10d\n", $file, $atime, $mtime;
  }
```

3. 这道题的解答是在前题基础上继续的。这里的解题关键是，用localtime把纪元时间转换成YYYY-MM-DD这样的日期格式。在把解答集成到完整程序之前，让我们

先来看看如何完成这个格式转换工作。假设时间戳放在$_里面（它是map的控制变量）。

我们从localtime文档查到相应字段的索引位置：

```
my( $year, $month, $day ) = (localtime)[5,4,3];
```

我们注意到，localtime返回的年份数字需要另外加上1900，月份数字需要另外加上1，才符合我们平时的习惯，所以另外做如下调整：

```
$year += 1900; $month += 1;
```

最后，再把这些数据组合成指定的格式，并对单个数字的月与日部分添上前置零：

```
sprintf '%4d-%02d-%02d', $year, $month, $day;
```

对一组时间戳进行这样的转换，只需用map依次变形。请注意，localtime默认不会使用$_变量，所以必须显式提供参数：

```
my @times = map {
  my( $year, $month, $day ) = (localtime($_))[5,4,3];
  $year += 1900; $month += 1;
  sprintf '%4d-%02d-%02d', $year, $month, $day;
  } @epoch_times;
```

这就是要替换之前程序中stat那行的代码，所以最终的结果应该是：

```
foreach my $file ( glob( '*' ) ) {
  my( $atime, $mtime ) = map {
    my( $year, $month, $day ) = (localtime($_))[5,4,3];
    $year  += 1900; $month += 1;
    sprintf '%4d-%02d-%02d', $year, $month, $day;
    } (stat $file)[8,9];

  printf "%-20s %10s %10s\n", $file, $atime, $mtime;
  }
```

设计本题的初衷在于引导读者使用第十七章提到的各个技术要点。当然除此之外还有其他实现方式，而且还更容易些。比如Perl自带的POSIX模块，它有一个strftime子程序，可以像sprintf那样把时间格式化为特定格式的字符串，它所接收的时间参数和localtime返回的内容完全一致，所以用map将它们联合起来变形的话，写起来更为简洁：

```
use POSIX qw(strftime);

foreach my $file ( glob( '*' ) ) {
  my( $atime, $mtime ) = map {
    strftime( '%Y-%m-%d', localtime($_) );
    } (stat $file)[8,9];

  printf "%-20s %10s %10s\n", $file, $atime, $mtime;
  }
```

附录 B

超越“小骆驼”

本书已经涵盖了很多知识点，但难免还有我们没有提及的内容。在这个附录里，我们将会多谈谈Perl能做什么，并提供深入了解Perl的参考资料。接下来要介绍的东西是崭新的，因此在你阅读本书时，可能部分内容已经有所变动。这也是为什么我们常常建议你自己查阅最新文档的原因。我们并不期望每位读者都仔细阅读本附录，但最好能快速浏览一遍标题，这样以后某人对你说“你根本不能在项目X里用Perl，因为Perl不能实现Y功能”时，你可以好整以暇地予以反击。

有一点很重要，这里提过之后接下来就不用反复提醒了：我们在此处没有介绍的其他重要特性会放到《Intermediate Perl》（也就是O'Reilly的“羊驼书”）中详加阐述。你绝对应该读一读“羊驼书”，尤其是在（独自或与他人合作）写出上百行程序代码时。没准你已经对Fred与Barney的故事感觉厌烦了，想要知道另外世界中的7个人[注1]在海难中漂流到荒岛上之后的求生故事。

在“羊驼书”之后，你还可以去看《Mastering Perl》（也就是O'Reilly的“小羊驼书”）。这本书涉及了日常Perl编程中遇到的问题，例如：性能检测和调试、配置文件、日志记录等等。还介绍了如何分析他人代码并最终将它们与自己的应用程序集成。

此外，还有许多很棒的书可以读。不过根据你所用的Perl版本的不同，请在花钱买书之前先看一下*perlfaq2*或者*perlbook*里面推荐的书目，免得买来不知所云或者内容过时的书。

注1：　可以把他们称为“漂流者（Castaways）”。

更多文档

Perl自带的文档乍看似乎浩如烟海，不过你可以用关键字在文档中搜索。搜索特定主题时，从*perltoc*（总目录）和*perlfaq*（常见问答集）这两节开始会比较好。在大部分的系统上，*perldoc*命令应能查到Perl核心包、已安装的模块以及相关程序的使用说明（包括*perldoc*本身）。也可以到*http://perldoc.perl.org*上在线阅读，虽然那里永远是最新版本的Perl文档。

正则表达式

没错，正则表达式的功能比我们所提到的还多。Jeffrey Friedl的著作《Mastering Regular Expressions》（O'Reilly）是我们读过的在这方面最出色的技术书籍之一[注2]。该书有一半内容是讨论一般的正则表达式，另一半内容则是讨论Perl的正则表达式，以及其他语言当中采用的兼容Perl正则表达式（Perl-Compatible Regular Expressions，简称为PCRE）的内容。该书深入介绍了正则表达式引擎内部的运作方式，并解释为什么某些模式的写法会比其他写法更好、更有效率。所有想要认真学习Perl的人都应该看看这本书。此外，也请参阅*perlre*文档（以及更新版本Perl中加入的*perlretut*和*perlrequick*文档）。当然，“羊驼书”和《Mastering Perl》也有很多关于正则表达式的内容。

包

包[注3]可以让你对源代码划分多个名字空间（namespace）。想象一下，有10个程序员正在合力开发某个大型项目。当你在开发这个项目时，使用了`$fred`、`@barney`、`%betty`、`&wilma`等全局变量，如果我不小心也用了这些变量名称，会有什么后果呢？包会让我们分别保存这些变量名称，我可以访问你的`$fred`，当然你同样也可以访问我所定义的这些变量而不会有意外发生。如果你想让Perl更灵活，使用包是必要的，它也可以让我们管理更大的程序。“羊驼书”对包也有详细的探讨。

注2：　这么说并不是因为出版商是O'Reilly，而是因为书确实很棒。

注3：　“包（package）”这个名字可能是个不幸的选择，它没有让人正确地联想到名字空间，而是让人想到打包的代码集（而这在Perl里其实是模块或库）。包其实是全局符号名的集合，把`$fred`或`&wilma`之类的东西收集在一起。而名字空间和代码包（chunk of code）是两个概念。

扩展Perl的功能

在Perl相关的论坛中常见的忠告之一就是："不要重新发明轮子。"你可以拿其他人已经写过的程序代码去使用。最常见的方式就是利用某个函数库或模块来扩展Perl的功能。有许多模块会跟着Perl一起被安装，至于其他模块则可以在CPAN找到。当然，你也可以编写一些属于自己的函数库或模块。

函数库

许多程序语言都提供了和Perl一样的函数库支持。函数库也就是为某个共同目标编写的子程序（subroutine）的集合。不过近来在Perl的使用上，使用模块的场合比使用函数库的多。

编写自己的模块

在一些少见的情况下，你也可能找不到需要的模块。此时，资深的程序员可以用Perl或其他程序语言（常常是C语言）写出一个新的模块。这部分的说明可以在*perlmod*和*perlmodlib*文档里找到。"羊驼书"也谈到了如何编写、测试以及发布模块。

数据库

如果你手上有一个数据库，Perl自然是可以访问和操作它的。这一节会介绍几种常见的数据库类型。我们之前已经在第十五章中看到过有关DBI模块的用法了。

直接访问系统数据库

Perl可以直接访问某些系统数据库，但有时需要借助一些相关的模块。比如Windows的注册表（Registry）数据库（记载了机器级的设定）、Unix的密码数据库（列出了用户名称与相关信息）以及域名数据库（将IP地址转换成机器名，或是做反向转换）。

访问平面文件数据库

如果想直接访问自己的平面文件数据库，也可以借助模块完成（似乎每一两个月就会出现新的模块，所以这里提供的列表迟早都会过时）。

其他操作符和函数

由于篇幅有限，我们无法介绍所有的操作符和函数：从..标量操作符到，标量操作符，从wantarray到goto（!），以及从caller到chr。请参阅*perlop*及*perlfunc*文档。

使用tr///进行转换

虽然`tr///`操作符看起来像正则表达式，但它实际上是把某组字符转换成另一组字符的操作符。它还可以迅速算出特定字符的出现次数。请参阅*perlop*文档。

Here文档

Here文档是一次引用多行字符串的好方法。请参阅*perldata*文档。

数学

Perl几乎能处理所有你能想象得到的任何数学计算。

高级数学函数

Perl内置所有基本的数学函数（平方根、余弦、对数、绝对值等），细节请参考*perlfunc*文档。虽然省略了某些函数（比如正切或以10为底的对数），但它们可以用现有的函数轻松组合而成，或是利用简单的模块完成（请参考`POSIX`模块，里面有许多常用的数学函数）。

虚数与复数

虽然虚数与复数不属于Perl的核心功能，不过可以借助模块来处理。这些模块能重载常见的操作符与函数，让你在处理复数时还能继续使用`*`做乘法，或用`sqrt`计算平方根。请参考`Math::Complex`模块。

超大与超高精度数字

如果需要，Perl可以处理任意大的数字，而且保持高精度。举例来说，你可以计算2 000的阶乘，或是计算π到小数点之后一万位。请参考`Math::BigInt`及`Math::BigFloat`模块。

列表与数组

Perl有许多特性，可以让人们非常容易地对整个列表或数组进行处理。

map与grep

在第十六章中，我们曾讨论过列表处理操作符`map`和`grep`。限于篇幅，我们无法介绍它们的全部功能。请参阅*perlfunc*文档，里面有进一步的信息及范例。也可以参考“羊驼书”

来获取更多使用map与grep的方式。

位与块

我们可以用vec操作符处理由位组成的数组，即位串（bitstring）。你可以设定第123位的值，清除第456位的值，或检查第789位的值。位串的大小没有上限。vec操作符的块长度可以设成2的乘幂，这在你需要把字符串视为由半字节（nybble）所组成的数组时很有用。请参阅*perlfunc*文档或者阅读《Mastering Perl》一书。

格式化

Perl的格式化功能可让你轻易制作出具有自动页首、格式固定的报表。事实上，这是Larry当初开发Perl的主要原因：作为实用摘录及报表语言。不幸的是它的功能十分有限。使用格式化最让人心碎的，莫过于发现格式化的功能无法满足日新月异的需求。如果真是如此，程序输出部分就得从头写起，换成使用格式化之外的方式。不过话又说回来了，如果你确定格式化能满足目前及未来的所有需求，它还是个相当酷的功能。请参阅*perlform*文档。

网络连接与进程间通信

Perl可以支持系统上运行的程序相互间彼此通信。本节展示几种常见方式。

System V进程间通信

Perl支持所有System V进程间通信的标准函数，包括：消息队列（message queue）、信号量（semaphore）、共享内存（shared memory）等。当然，Perl里的数组和C语言不同，并不是存储在连续的内存块中[注2]，因此共享内存无法直接共享Perl数据。不过，有些模块可以帮忙转译数据，让Perl的数据看起来像存储在共享内存里。请参阅*perlfunc*和*perlipc*文档。

套接字

Perl对TCP/IP套接字（socket）提供了完整的支持，所以只要用Perl就可以写出Web服务器、Web浏览器、Usenet新闻服务器或客户端、finger服务器或客户端、FTP服务器或客户端、SMTP/POP/SOAP的服务器或客户端以及Internet上所用的任何协议的客户端或服

注2： 事实上，说 Perl 的数组存储在“一块内存”里面通常是不对的。它们几乎一定会分散在许多小块的内存中。

务器。你可以在Net::名字空间里找到很多实现这类协议的模块，并且有许多还是Perl自带的。

当然，你不需要了解底层技术细节，因为所有常见的协议都已经有模块可用了。比如，你只要用LWP模块再加上一两行程序代码，就能做出Web服务器或客户端[注3]。如果想借鉴高质量的程序代码，不妨参照一下LWP模块。事实上，它是一系列互相支持的模块，它已经实现出了Web上绝大部分的功能。至于其他协议，请用协议名称来搜索相关的模块。

安全

Perl提供了不少与安全有关的强大功能，这让Perl程序比相应的C程序更加安全。其中最重要的可能就是一般称为污染检查（taint checking）的数据流分析。当它被启用时，Perl会记住哪些数据是来自用户或环境的（因此不值得信赖）。一般来说，当这些所谓“受污染的（tainted）”数据对其他进程、文件或目录产生影响时，Perl会禁止该项操作并中断程序。这并不完美，但却能有效避免安全相关的问题。还有在这里讲不完的细节，请参考*perlsec*文档。

程序调试

Perl自带了一个很棒的调试器，它支持断点（breakpoint）、观察点（watchpoint）、单步执行（single-stepping）以及所有命令行调试器该有的功能。它其实是用Perl写成的，如果它本身有缺陷，我们也不知道该如何对它进行调试。除了基本的调试器命令之外，你还可以在程序运行到一半的时候从调试器来运行Perl程序代码——调用子程序、改变变量甚至是重新定义子程序。最新的信息，请参考*perldebug*文档。另外“羊驼书”对调试器的使用作了详细介绍。

另一个调试技巧就是使用B::Lint模块。这个模块能对潜在的问题发出警告，而这些问题可能即使打开了-w开关都不会发现。

注3：　虽然借助LWP模块可以很轻松地实现可下载页面或图像的“Web浏览器”，但如何实际显示给用户看又是另一回事。你可以利用Tk或Gtk插件来驱动X11进行显示，或使用Curses模块在字符终端上绘图。别无他法，从CPAN下载并安装合适的模块就是了。

命令行选项

Perl有许多不同的命令行选项，其中有不少选项能让你直接从命令行写出有用的程序。请参阅*perlrun*文档。

内置变量

Perl有一大堆内置变量（比如@ARGV和$0），它们能提供非常有用的信息或者帮助控制Perl的运作方式。请参阅*perlvar*文档。

语法扩展

Perl语法还有更多技巧，包括continue块与BEGIN块。请参阅*perlsyn*和*perlmod*文档。

引用

Perl的引用（reference）跟C语言的指针（pointer）差不多，不过工作原理比较类似Pascal或Ada里的相应功能。引用会“指向”某个内存位置，但因为没有指针运算和内存的直接分配、释放功能，所以你可以确定任何引用都是有效的。引用可以用来实现面向对象程序设计、复杂的数据结构以及其他有用的技巧。请参阅*perlreftut*和*perlref*文档。“羊驼书”里有详细讨论引用的内容。

复杂的数据结构

引用能够让我们用Perl建立复杂的数据结构。比如需要二维数组的话，就可以通过数组的引用来构造。[注3] Perl还能构造更加有趣的数据结构，比如由哈希组成的数组、由哈希组成的哈希、由哈希的数组组成的哈希等等[注4]。请参阅*perldsc*（数据结构范例）*perllol*（由列表组成的列表）这两个文档。“羊驼书”详细介绍了这部分内容，包括复杂数据的操作技巧，比如排序和统计等。

面向对象程序设计

没错，Perl也有对象，并且和其他语言是术语兼容（buzzword-compatible）的。面向对象（object-oriented，简称OO）程序设计让你能够通过继承（inheritance）、覆盖

注3：　其实严格来说并不完全这样，不过你可以装得非常像，直到自己都不太记得有什么差别。

注4：　事实上，这些东西Perl都做不到，它们不过是简称而已，实际情况并非如此。我们所谓的“数组的数组”，在Perl里面其实是“存放数组引用的数组”。

（overriding）以及动态方法判定（dynamic method lookup）来自行定义功能相关的数据类型 [注5] 。但不像其他编程语言，Perl并不要求你一定要用面向对象的方式编写程序。

不过，如果程序代码长度大于N行，那么以面向对象的方式设计对程序员而言可能比较有效率（虽然运行时可能会慢一点点）。没有人知道N的值确切是多少，不过我们估计它在几千左右。请阅读文档*perlobj*及*perlboot*作为入门。进一步的权威信息请参考由Damian Conway编写的《Object-Oriented Perl》（Manning Press）这本好书。“羊驼书”也同样详细介绍了有关对象的内容。

在我们更新本书的时候，基于Moose的元对象（meta-object）系统正在变得非常流行。它是在原始粗放式Perl对象实现的基础上，按照更切合面向对象编程理念的方式构建起来的，并且提供了更富语义和逻辑的使用界面。

匿名子程序和闭包

乍听之下也许很奇怪，不过没有名称的子程序也是很有用处的。这种子程序可以作为其他子程序的参数，也可以放进数组或哈希里作为跳转表(jump table)来使用。闭包（closure）则是从Lisp世界引入的概念，它的意思（差不多）是具有私有数据的匿名子程序。照例，“羊驼书”和《Mastering Perl》里也谈到了这方面的内容。

捆绑变量

捆绑变量（tied variable）可以用惯常的方式来访问，但每种访问方式的背后使用的是你自己的程序代码。所以，你可以建立存储在远程机器上的标量或某个总能保持排序的数组。请参阅*perltie*文档或者《Mastering Perl》。

操作符的重载

你可以利用overload模块重新定义某些操作符，包括加法、连接、比较甚至是从字符串到数字的自动转换。比如，实现复数的模块就可以用这种方式来让复数乘以8得出正确的复数。

注5：　面向对象有专门的行话。事实上，任何两种面向对象语言中的同一个术语也常常会在细节上有所不同。

动态加载

动态加载就是让程序在运行过程中加载某些额外模块的功能，动态加载后继续运行后面的程序。动态加载Perl程序代码从来不是问题，不过有意思的是，Perl还能动态加载二进制扩展模块[注6]。用非Perl语言写出来的Perl模块就是这样做出来的。

嵌入

（从某种角度来说）与动态加载相对的技术就是嵌入了。

如果你想写个非常酷的文字处理器，（假设）开始时是用C++语言实现的[注7]。现在，你希望用户能够使用Perl的正则表达式来执行强大的“查找并替换”功能，于是将一段Perl程序嵌入你的程序里。接下来，你也可以将Perl的某些功能开放给用户。高级用户可以用Perl编写子程序，使之成为菜单里的功能。他们也可以写一小段Perl程序来自定义文字处理器的操作。之后，你在网站上留了一个空间，让用户分享并交流这些Perl程序片段。这样一来，就会有上千名程序员在扩展此程序的功能，而你的公司不必为此有额外支出。为了这些好处，你要付钱给Larry吗？完全免费，请参考Perl所附的授权条款。Larry实在是个大好人，你至少该寄给他一封感谢函吧。

虽然我们还没听说过这种文字处理器，但有些人已经使用同样的技巧做出功能强大的其他程序了。其中一个例子是Apache的`mod_perl`，它将Perl嵌入到功能已经十分强大的Apache Web服务器里面。如果你也想要嵌入Perl到其他语言编写的项目中去，不妨参考一下`mod_perl`的做法，因为它是完全开源的，所以可以研究一下它是怎么做的。

把其他语言所写的程序转换成Perl程序

如果你有以*sed*或*awk*语言写成的程序，却希望能用Perl做同样的事，那你大可放心。Perl不但具备这两个语言的所有功能，还附带转换程序，它们大概已经安装到你的系统上了。请参考*s2p*（从*sed*转换到Perl）及*a2p*（从*awk*转换到Perl）的说明文档[注8]。因

注6：只要系统支持，通常都能以动态加载的方式进行二进制扩展。如果系统不支持，也可以对扩展模块进行静态编译，也就是说Perl在编译时即包含了该扩展模块，可以直接使用。

注7：没准用这个语言来写文字处理器是很合适的。没错，我们喜爱Perl，但并没有发毒誓不再用别的语言。如果X语言是最佳选择的话，就该使用它。只不过X常常是Perl罢了。

注8：如果你在使用*gawk*、*nawk*或其他衍生版本，*a2p*也许就无法进行转换了。这里所提到的两个转换程序都是在很久之前写成的，除了让它在新版的Perl中继续运行之外，它们几乎没被更新过。

为机器写的程序比不上人写的，所以用这种方式产生的代码并不怎么样，但它至少是个开始，在它基础上做进一步修改也很容易。转换后的程序的运行速度可能有所不同。不过，在修复好程序代码里明显的瓶颈之后，应该和原来的程序差不多。

希望在Perl里使用已经用C语言实现了的算法吗？也请放心，将C程序代码编译成Perl模块并不难。事实上，任何能编译成目标码的程序语言一般也都能被转换成Perl模块。请参考*perlxs*文档、Inline模块以及SWIG系统。

你想将现有的shell脚本转换成Perl吗？真不幸，并没有从shell自动转换到Perl的方法。这是因为shell本身几乎什么事都不做，它大部分的时间都在运行别的程序。当然，我们可以写个程序来将shell里的每一行都转成system调用，可这样会比shell直接调用慢得多。要将shell里所用到的*cut*、*rm*、*sed*、*awk*和*grep*转换成有效率的Perl程序代码，实在需要人类程度的智力才办得到。将shell脚本从头改写是比较好的做法。

把find命令行转换成Perl程序

系统管理员的常见任务之一就是递归搜索目录树来寻找某些特定文件或目录。在Unix上，这通常要靠*find*命令来完成。这件事，我们也可以直接用Perl来达成。

Perl自带的*find2perl*命令所接受的参数和*find*命令的相同。不过，*find2perl*并不会执行搜索操作，而是输出用于进行搜索的Perl程序源代码。既然是程序代码，你就可以根据需要自行修改（虽然程序的风格会有点奇怪）。

*find2perl*有个很好用的参数是*find*没有的，那就是-eval选项。它后面的参数是实际的Perl程序代码，这段代码会在每次文件被找到时运行。当它运行时，当前工作目录将会是该文件所在的目录，$_的值则是找到的条目名。

下面是*find2perl*的一种用法。假设你是Unix系统上的管理员，想要将*/tmp*目录下陈旧的文件全部删除[注9]。下面就是生成该程序的命令：

```
$ find2perl /tmp -atime +14 -eval unlink >Perl-program
```

该命令会在*/tmp*（及其所有子目录）里寻找atime（最后访问时间）离现在至少14天的所有条目。它会对每个条目运行Perl的unlink代码，而且会将默认变量$_的值当成要删除的文件名。输出结果（重定向到*Perl-program*这个文件里）将会是执行上述任务的程序代码。接下来，只要安排它在适当时间运行就行了。

注9：　这通常是在每天清晨由*cron*定时完成的任务。

让你的程序支持命令行选项

如果想让你的程序支持命令行选项（就像Perl自己的-w警告选项之类的），可以选用现成的模块并按照标准惯用方式实现。请参考Getopt::Long及Getopt::Std模块的说明文档。

嵌入式文档

Perl自己的说明文档是以*pod*（plain-old documentation）方式写成的。你可以将这种文档直接嵌到程序代码内，随后在需要时可以把它转换成纯文本、HTML或许多其他格式。请参考*perlpod*文档。“羊驼书”也介绍了这部分内容。

打开文件句柄的其他方式

打开文件句柄时还有许多别的模式可供使用。请参阅*perlopentut*文档。Perl内置的open的功能十分完备，所以它有一份专门的说明文档。

线程与派生进程

Perl现在已经支持线程了。虽然（在本书编写时）线程功能还是实验性的，但在某些应用中已经非常好用了。Perl对fork的支持（当然是在支持该功能的操作系统上）比较完善。请参阅文档*perlfork*和*perlthrtut*。

图形用户界面（GUI）

Perl支持许多GUI工具包。请到CPAN查阅Tk、Wx等模块的文档。

更多……

只要看看CPAN上的模块列表，你就可以找到各种不同用途的模块：从产生图表及图像，到下载电子邮件；从计算贷款分期付款，到预测日落时间。新的模块不断涌现，所以你现在看到的Perl又比我们写本书时强大多了。我们不可能一直跟上新的模块，所以就到此为止。

由于Perl的世界不断扩张，所以连Larry也自认无法跟上Perl的所有进展。他并不会对Perl感到无聊，他总是能够在这个不断扩张中的世界里找到新的角落。本书的作者大概也有同感。Larry，谢谢你！

附录C

Unicode入门

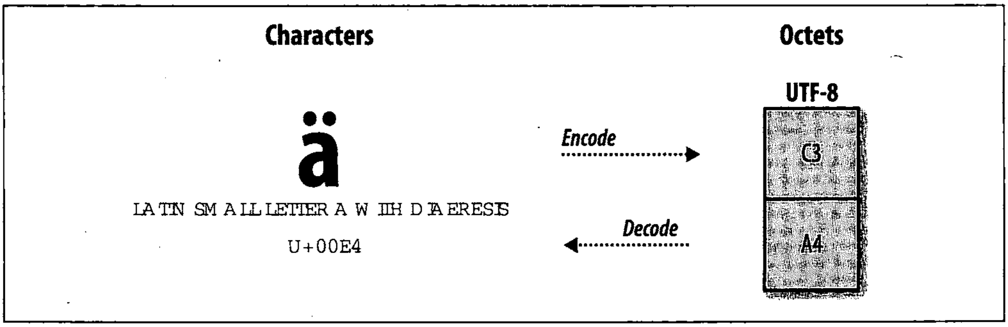
本附录并非关于Unicode的完整介绍，我们只是为了帮助你在学习本书时能理解Unicode相关部分的内容。之所以说Unicode有点难，不光是因为它对于字符串有了全新的理解，有大量调整的词汇表，也因为历史原因，很多计算机语言对它的支持一直不够完善。Perl 5.14在Unicode方面已经有了非常大的改善，虽然还不是十分完美，不过也算是你能找得到的最好的对Unicode的支持了。

Unicode

Unicode字符集（Unicode Character Set，简称UCS）是字符（*character*）到代码点（*code point*）的抽象关系映射。它和字符在内存中的特定表示方法没有关系，也就是说，我们在谈论字符时可以不用考虑操作系统的区别，它们在任何一个平台上都总是同一个实体。而我们讲的编码（*encode*），是指将字符的代码点按照特定形式存储到内存中的方式，由它建立起抽象字符映射和计算机物理实现之间的桥梁。你可能认为谈到存储时应该使用“字节（byte）”这个术语，不过在我们谈论Unicode时应该用术语位组（*octet*）（参见图C-1）。不同的编码方式存储位组的方式不同，从另一个意义上来说，反过来把特定位组按照既定的编码方式转换成字符的过程就称为解码（*decode*）。其实你不用担心这些底层细节，Perl已经帮你处理好所有这些内部事务了。

当我们谈到代码点的时候，我们会用十六进制数字表示，如：(U+0158)，这个代码点表示字符Ř。代码点还有名称，像这个字符的名称就是“LATIN CAPITAL LETTER R WITH CARON”。不止如此，代码点还能判断知晓有关它本身的一些信息。它知道自己是否为大写字符还是小写字符，是字母还是数字，或者属于空格一类的空白符等等。并

且如果存在的话，它还知道对应于自己的大写版本、标题版本（title case）、小写版本是什么字符。所以在Unicode里面，我们不光能处理单个字符，还能同时处理一组特定类型的字符。所有这些都定义在*perl*自带的Unicode数据文件中。到Perl的库目录中找找看名为*unicore*的目录，Perl就是依赖它完成各种Unicode字符操作的。



图C-1：代码点只是字符代号，并非存储内容。编码就是把字符转换为存储位组的过程。

UTF-8和它的朋友们

Perl里面推荐使用的是UTF-8编码，它是“UCS Transformation Format 8-bit”的简短说法。这个编码的定义，是某天晚上Rob Pike和Ken Thompson在新泽西州共进晚餐时写在餐垫背面的[注1]。这只是Unicode编码的一种实现方式，但由于它没有其他编码方式的种种缺陷，所以变得极为流行。不过要是你用Windows的话，最好选用UTF-16编码。对于这种编码，我们没什么特别想说的，那就遵从妈妈的“沉默是金”的教导好了。

让所有参与者达成共识

要让所有工具和系统都设置成使用Unicode往往并不那么简单，因为每一个使用到Unicode字符的系统都需要知道原来用的是什么编码方式，才能按照这种方式正确显示或者处理。任何一个环节的步调不一致，都会产生类似乱码那样的效果，同时又很难推断具体问题出在哪个环节。如果程序输出的是UTF-8编码的字符，那么显示这个结果的终端程序必须知道这一点才能正确显示。如果给Perl程序的输入数据是UTF-8编码的字符，那么Perl程序也必须知道这一点，才能对输入的字符串作出正确的解析和处理。如果存放UTF-8数据到数据库，那么数据库也必须知道这一点，在保存和提取数据时都按照这个编码方式进行操作。在编写Perl程序时，如果希望*perl*解释器把源代码中的字符当

注1：　不妨读一读Rob Pike自己在*http://www.cl.cam.ac.uk/~mgk25/ucs/utf-8-history.txt*上讲的关于发明UTF-8编码的故事。

作UTF-8编码的字符来解析的话，也同样需要设定编辑软件，让其按照UTF-8编码的方式保存程序文件。

我们不知道你正在用的是哪一款终端程序，我们也不打算把所有终端软件的配置方法全都列在这里。不过对大多数终端程序来说，试试看到偏好设置里找一下有关编码方面的设定，多半都很简单。

除了对终端的编码设置，各种程序也需要知道最终放在终端里显示的输出该用什么编码。大多数程序会参考名称以LC_*开头的环境变量，也有一些会参考自己事先约定名称的环境变量：

```
LESSCHARSET=utf-8
LC_ALL=en_US.UTF-8
```

如果输出内容过长，使用分页程序（比如less、more、type等）显示却发现字符显示不正常的话，请阅读它们各自的文档，看看应该如何设定才能让它们知道正确的字符编码。

谜样的字符

如果长久以来你都习惯用ASCII编码看待字符的话，那么现在改用Unicode就要换换脑筋了。举个例子，请问é和é有什么差别？光是看恐怕很难说出不同来，就算你读的是本书电子版也很难判断究竟有何不同，说不定在本书出版过程中不小心“修掉了”它们之间的差别。你甚至可能不相信这里有所不同，不过事实确实如此。前者是单个字符，而后者却是两个字符。对人来说，它们都表示相同的字素（*grapheme*），或者说字形（*glyph*），不管计算机是用什么方式表示这个字素的，但最终指代的都是同一个东西。我们通常只关心最终这个显示出来的字素，它才是最终影响读者并对读者产生意义的东西。

在Unicode出现以前，常见的字符集会把é这样的字素定义为原子（atom），或者说单个实体（single entity），就比如前面这段文字中的第一个字符示例（相信我们）。不过，Unicode还有一个表示字素的方式，就是使用表示读音或其他衍生注解意义的记号（*mark*）字符和一个普通的非记号（*nonmark*）字符组合而成。第二个字符示例é就是由非记号字符*e*（U+0065，LATIN SMALL LETTERE）和一个表示字母上方的小撇的记号字符 ′ （U+0301，COMBINING ACUTE ACCENT）组成。这两个字符共同构成了最终我们看到的字素。事实上，这也正是我们不能一概称之为字符而要改称为字素的原因。同一个字素的构成方式可能不止一种，也许只用了一个字符，也许组合了多个字符。可能有点过于学究气了，不过这里讲的都是基础，理解了这些才能更好地理解接下来对Unicode的处理，知其所以然。

如果世界可以从头来过，Unicode也许不必同时维护单个字符版本的é字符，毕竟组合的方式更加灵活。但出于历史原因，单个字符的版本早已存在，Unicode不得不考虑向后兼容，并采取一切努力善待这类字符。对ASCII编码来讲，它的字符序号和Unicode的字符代码点完全一致，都是从0到255的那一段。所以，把ASCII字符串当作UTF-8编码来处理完全没事，不过在UTF-16里就不是这样了，这种编码对所有字符都采取两个字节的形式定义。

单个字符版本的é可以称作为组合（*composed*）字符，因为它是用一个代码点表示两个或多个字符。它把非记号字符和记号字符合并成单个字符（U+00E9，LATIN SMALL LETTER E WITH ACUTE），最终只用一个代码点来表示。而表示同一个字素的两个字符则构成了相应的分解（*decomposed*）版本。

那么，为何我们在意这两种不同的表示方法呢？要是用不同的字符表示同一个字素，我们该怎么正确按照实际的字素意义来排序呢？Perl的`sort`函数关心的仅仅是字符层面上的事，而不会过问字素，所以`"\x{E9}"`和`"\x{65}\x{301}"`这两个逻辑上都表示é的字符串，是不会在排序结果中排在相邻位置上的。而我们总是期望按é字素的意义排序，不管它到底是用哪种方式表示的，组合字符也好，分解字符也好，都一样。但计算机看到的只是字符，它并不知道其中的意义，所以不会按照人们需要的方式进行排序。我们马上向你展示解决方案，另外不妨请你一并读下第十四章。

更加谜样的字符

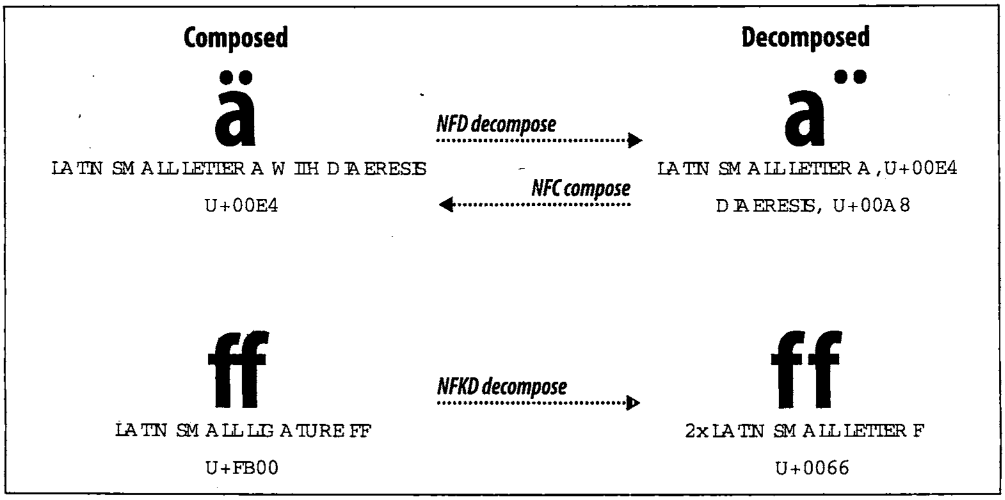
事情还可以变得更为离奇，虽然大多数人不会特别在意。请问，*fi*和*ﬁ*之间又有什么差别呢？只要排字员没有对此进行“优化”，第一个其实是由分开的*f*和*i*组成的，第二个是两者合在一起构成的连字（*ligature*），一般定义为字素，方便读者辨识 [注2] 。字符*f*上面的部分看起来好像要强加在字符*i*上面的点所在的私人空间里一样，看起来有点丑陋 [注3] 。你可能从未注意到这些细节，不过你可以在本段找到几个例子，如果你读的是关于字体编排的书籍，这种情况应该会更多一些 [注4] 。

注2：　O’Reilly 的自动排版系统不会把我们的 *ﬁ* 转换为对应的连字，我们只能自己手工键入。不加转换的文档流转工作可能处理起来要快些（faﬅer），而与此同时，我们在普通正文中混杂（shuﬄe）一些连字也不会有什么问题。（译注：为了印证作者所言非虚，原文faﬅer 和 shuﬄe 这两个词，暗暗使用了形式相近的连字ﬅ和ﬄ代替中间的部分。）

注3：　我们识别文字的时候并不是按照每一个字母来看的，而是把它们作为一个整体，所以合并后的连字能稍稍方便我们的模式识别。基于这样的原因，字体设计师把这样的两个字素合并为一个连字字素。

注4：　但不是电子书。好像电子书一般都不太在意字体是否漂亮。

这和é的组合形式、分解形式相类似。é的两种表示方法实际上可说是完全等价（*canonically equivalent*）的，因为不管如何构造这个字素，最终的视觉呈现和字素意义都是相同的。但*ﬁ*和*fi*的视觉呈现并不相同，所以它们只是不完全等价（*compatibility equivalent*）的[注5]。不管是完全等价还是不完全等价，我们都不需要关心如何分解为方便排序的普通形式（见图C-2）。



图C-2：我们可以分解并重新组合完全等价的字素表示形式，但只能分解不完全等价的字素表示形式。

比如你想要检查字符串中是否包含é或者*ﬁ*，但并不关心原始用的是哪种字符表示形式。首先，需要将字符串分解为普通形式，这好比是在做归一化。要分解Unicode字符串，得用Perl自带的`Unicode::Normalize`模块，它提供了两个分解字符串的函数。你可以用NFD（*Normalization Form Decomposition*）子例程把完全等价形式的字素转换为对应的分解形式。另外，也可以用NFKD（*Normalization Form Kompatibility Decomposition*）子例程分解为不完全等价的字素。下面例子中的字符串包含了组合字符，之后再以不同方式分解后匹配。包含“oops”字眼的消息应该不会输出，而包含“yay”字眼的会在匹配后打印输出：

```
use utf8;
use Unicode::Normalize;

# U+FB01        - ﬁ 连字
# U+0065 U+0301 - 分解形式的 é
```

注5：　参考《Unicode Standard Annex #15》里面关于“Unicode Normalization Forms”的细节阐述。

```
# U+00E9       - 组合形式的 é

binmode STDOUT, ':utf8';

my $string =
    "Can you \x{FB01}nd my r\x{E9}sum\x{E9}?";

if( $string =~ /\x{65}\x{301}/ ) {
    print "Oops! Matched a decomposed é\n";
}
if( $string =~ /\x{E9}/ ) {
    print "Yay! Matched a composed é\n";
}

my $nfd  = NFD( $string );
if( $nfd =~ /\x{E9}/ ) {
    print "Oops! Matched a composed é\n";
}
if( $nfd =~ /fi/ ) {
    print "Oops! Matched a decomposed fi\n";
}

my $nfkd = NFKD( $string );
if( $string =~ /fi/ ) {
    print "Oops! Matched a decomposed fi\n";
}
if( $nfkd =~ /fi/ ) {
    print "Yay! Matched a decomposed fi\n";
}
if( $nfkd =~ /\x{65}\x{301}/ ) {
    print "Yay! Matched a decomposed é\n";
}
```

你应该看到了，NFKD方式总是能匹配分解形式的字符，因为NFKD()既能分解完全等价形式，也能分解不完全等价形式。而NFD方式则无法处理不完全等价形式：

```
Yay! Matched a composed é
Yay! Matched a decomposed fi
Yay! Matched a decomposed é
```

所以这里提示我们需要注意的是：你可以对完全等价形式分解或重新组合，但却不能对不完全等价形式重新组合。如果把连字*ﬁ*分解开来，会得到两个独立的字素f和i，但反过来，重新组合时谁也不能断定这两个连在一起的字符的本意是某个字素，还是原本就是分开独立的两个字符[注6]。这也就是说，完全等价和不完全等价的不同之处在于：完全等价的分解和组合形式看起来总是相同的。

注6：这就是为什么我们不谈NFC和NFKC的原因。这两种方式会先做分解，再做重新组合的操作，但NFKC未必总能回到分解前原来的形式。

在Perl中处理Unicode

本节是关于Perl程序中Unicode常见用法的快速小结。这并不是完整指南，甚至有些细节是我们故意略去不讲的。关于Unicode的处理实际上是一个很大的主题，我们并不想在一开始就吓到你。请先从本附录开始，学一点基本的东西，到碰到实际问题时，再求助于本附录末尾列出的文档。

在源代码中使用Unicode

如果你想要在编写源代码时使用UTF-8字符，需要告诉*perl*解释器你的程序文件是以UTF-8编码的。只要打开utf8编译指令就可以了，它的唯一任务就是告诉*perl*按照正确方式解析源文件。比如下面代码中的字符串就用到了若干Unicode字符：

```
use utf8;

my $string = "Here is my ? résumé";
```

甚至还可以在变量名或者子程序名中使用Unicode字符：

```
use utf8;

my %résumés = qw(
    Fred => 'fred.doc',
    ...
    );

sub π () { 3.14159 }
```

utf8编译指令的唯一任务就是告诉*perl*解释器按照UTF-8编码，方式解析源代码。它不会帮你做其他任何辅助性的工作。只要你决定开始用Unicode工作，那就最好一直留着这个编译指令，除非有很好的理由不这么做。

谜样字符也有名字

除了代码点外，Unicode字符还有属于自己的名字。如果要输入的字符很难通过键盘键入，又不记得对应的代码点，那就用它的名字好了（虽然要输入的内容也不少）。Perl自带的charnames模块就是用来支持通过名字表示Unicode字符的。只要在双引号上下文中把名字放在\N{...}里面：

```
my $string = "\N{THAI CHARACTER KHOMUT}"; # 代码点为 U+0E5B
```

可能你注意到了，在匹配和替换操作的模式部分，也是使用类似的双引号上下文，但我们还有一个名为\N的字符集合简写，表示除换行符以外的所有字符（参见第八章）。虽

然这里的形式相近，但一般不会出现歧义，所以就算名字相同也没关系，Perl能够分清楚[注7]。

从STDIN读并写到STDOUT或STDERR

在计算机的底层，输入和输出的都只是位组。你的程序需要知道用什么编码方式对这些位组进行编码或解码。我们之前在第五章中已经谈了很多，这里做一下小结。

对文件句柄来说，有两种指定编码方式的办法。第一个是使用binmode操作符：

```
binmode STDOUT, ':encoding(UTF-8)';
binmode $fh, ':encoding(UTF-16LE)';
```

也可以在打开文件句柄时在模式部分指定：

```
open my $fh, '>:encoding(UTF-8)', $filename;
```

如果要对所有将要打开的文件句柄设定默认编码方式，可以用open编译指令指定。我们可以把所有输入和输出的句柄都设为UTF-8编码的：

```
use open IN  => ':encoding(UTF-8)';
use open OUT => ':encoding(UTF-8)';
```

也可以用一条语句分别设定：

```
use open IN  => ":crlf", OUT => ":bytes";
```

如果希望输入和输出使用同样的编码方式，可以像下面这样同时指定，这里的IO写上或省略都可以：

```
use open IO  => ":encoding(iso?8859?1)";
use open ':encoding(UTF-8)';
```

因为默认文件句柄已经处于打开状态，所以需要使用:std子编译指令来让它们改用之前设定的编码方式：

```
use open ':std';
```

如果之前没有显式定义过使用何种编码方式，上面这条语句是不会起任何作用的。

你也可以在命令行使用-C开关，通过后面相应的参数指定应用到标准文件句柄上的编码方式：

注7：　有关于\N的进一步详细讨论，请参阅*http://www.effectiveperlprogramming.com/blog/972*。

```
I     1   STDIN 假定为用 UTF-8
O     2   STDOUT 将会用 UTF-8
E     4   STDERR 将会用 UTF-8
S     7   I + O + E
i     8   输入数据流的默认 PerlIO 层会使用 UTF-8
o    16   输出数据流的默认 PerlIO 层会使用 UTF-8
D    24   i + o
```

参阅*perlrun*文档以获取有关命令行开关的详细信息，包括这里的-C的用法。

读写文件

我们之前在第五章讨论过文件读写方面的内容，这里略做小结。在打开文件时，使用三项参数形式的写法，就能明确指定按照哪种编码方式读写文件：

```
open my( $read_fh ),   '<:encoding(UTF-8)',  $filename;
open my( $write_fh ),  '>:encoding(UTF-8)',  $file_name;
open my( $append_fh ), '>>:encoding(UTF-8)', $file_name;
```

不过要记住，输入数据的编码方式并不是你决定的（至少不是在你的程序内部指定的）。除非你明确知道来源数据所使用的编码方式，否则不要指定输入句柄的编码设定。要知道，就算不作声明，实际上还是会用:encoding做解码工作的。

如果不知道输入数据采用何种编码方式，或者根本就无法预测的话（按编程法则的说法来看，只要运行足够多次，就能得到所有可能正确的编码），不妨观察一下原始数据流中的数据，猜一个合乎情理的编码试试看，或者交给Encode::Guess模块来猜。此外还有一些取巧的办法，不过这里我们不再赘言。

在数据进入程序之后，你就不必担心编码之类的事情了。Perl会聪明地处理有关内存存储和操作的事宜。一直到你把数据写到文件（或者发送到某个套接字，或者其他什么输出句柄），才需要再次指定编码方式对数据进行编码。

处理命令行参数

正如我们之前所说的，如果要把手上的数据当作Unicode来处理的话，需要留心数据来源。特殊变量@ARGV就是一个特例，它的值取自命令行参数，而命令行参数在输入时参考的是本地环境设置，所以需要取得该设置并按其编码方式解码：

```
use I18N::Langinfo qw(langinfo CODESET);
use Encode qw(decode);

my $codeset = langinfo(CODESET);

foreach my $arg ( @ARGV ) {
    push @new_ARGV, decode $codeset, $_;
```

```
}
```

处理数据库

编辑跟我们说，都到本书结尾了，要是连这个话题都要谈，一定会超出篇幅预算的！我们也知道，这么有限的篇幅很难面面俱到，不过没关系，这里要说的其实不光和Perl有关。于是他说那就稍微讲几句吧。说真的，有那么多的数据库服务器，对数据的处理方式又各不相同，我们无法一一道来，实在遗憾。

不管怎么说，有些数据最终还是要存到某个数据库的。Perl里面最流行的访问数据库的模块是`DBI`，它可以透明处理Unicode字符，也就是说，它不加干涉地直接把原始数据递交给数据库服务器存储。每种数据库都需要一个对应的驱动器模块（比如`DBD::mysql`模块），请查阅这类模块的文档看看需要做哪些编码方面的设定。另外，你还要对数据库的工作方式、表结构以及特定字段等分别做出正确的编码设定。现在你明白为什么这个话题会超出篇幅预算了吧！

进阶阅读

Perl文档中有许多谈论Unicode的部分，包括*perlunicode*、*perlunifaq*、*perluniintro*、*perluniprops*、*perlunitut*文档等。另外，别忘了查阅你正在使用的Unicode模块的文档。

位于*http://www.unicode.org*的Unicode官方站点基本上涵盖了各种你所感兴趣的有关Unicode的内容，它也是一个很好的学习起点。

本书作者之一另著有《Effective Perl Programming》（Addison-Wesley）一书，其中专门有一章内容讨论Unicode方面的问题。（译注：该书第二版的中文版已经上市，书名为《Perl高效编程》。）

大家网
TopSage.com

作者简介

Randal L. Schwartz

Randal L. Schwartz已经是软件行业历练了二十多年的老手了，他在软件设计、系统管理、系统安全、技术写作和培训等方面拥有丰富的经验。Randal参与编著的“必读”书籍有：《Programming Perl》《Learning Perl》以及《Learning Perl onWin32 Systems》等（全部由 O’Reilly出版），另外还著有《Effective Perl Programming》（由Addison-Wesley出版）。（译注：该书第二版已由人民邮电出版社于 2011 年发行简体中文版，书名为《Perl高效编程》。）他还是《WebTechniques》《PerformanceComputing》《SysAdmin》以及《Linux Magazine》等杂志的 Perl 专栏作家。

不仅如此，他还是 Perl 新闻组的热心奉献者，从 comp.lang.perl.announce创建伊始就负责协助管理大小事务。他以风趣的言谈和扎实的技术功底，赢得了圈内的普遍赞誉（虽然有些传奇故事是他自己爆出来的也说不定）。Randal总是想着回报 Perl 社区赋予他的一切，于是着手参与筹建 Perl Institute 基金。他还是 Perl Mogers(perl.org) 董事会成员，该机构是全世界范围内 Perl 开发者一致拥护的社团组织。从 1985 年起，Randal 拥有了自己运营的Stonehenge Consulting Services 公司。可以发送邮件到 merlyn@stonehenge.com和Randal 聊聊有关 Perl 方面的话题。

brian d foy

brian d foy 是一位多产的 Perl 培训师和技术写手，他运作的 The Perl Review旨在通过培训教育、技术咨询、代码审校等方式，帮助人们理解 Perl 的方方面面。他还是 Perl技术大会的常客，他参与编著的书籍有《Learning Perl》《Intermediate Perl》以及《Effective Perl Programming》（由 Addison-Wesley出版），而他自己编著的有《Mastering Perl》。他在 1998 到2009 年间担任 Stonehenge Consulting Services 公司的讲师，而在他读物理学研究生时就已经是一名 Perl用户了，从拥有第一台计算机开始便是一名 Mac 死忠用户。他创办了第一个 Perl 用户社群，即 New York Perl Mongers，继而又创办了非盈利性的组织 Perl Mongers 公司，以协助全球超过 200 个 Perl用户社群能够顺利发展。他还负责维护核心 Perl 文档中的 perlfaq 部分以及分享在 CPAN 上的诸多模块和独立脚本。

Tom Phoenix

Tom Phoenix 自 1982 年起就开始投身教育事业。13年来他在科学博物馆工作时多半与解剖、爆炸为伍，摆弄过高压电，接触过有趣的动物。1996 年起，他开始了在 Stonehenge Consulting Services 公司的 Perl 教学生涯。因为工作关系，他到处旅行，所以要是你在

当地的 Perl Mongers大会上碰巧遇上他也不用太过惊奇。只要时间允许，他会到 Usenet 的 comp.lang.perl.misc 和comp.lang.perl.moderated 新闻组回答问题，或者对 Perl 的开发工作积极进言。在工作上他是 Perl专家、黑客，在生活上他也乐于投入时间摆弄密码学，说说西班牙文。他目前定居在美国俄勒冈州波特兰市。

封面介绍

《Perl语言入门》第六版的封面动物是骆马（*Lama glama*）。它是骆驼（camel）的同类，原生于安第斯（Andean）山脉附近。骆马类族群里还包括可驯养的羊驼（alpaca），以及它野生的祖先原驼（guanaco）和小羊驼（vicuña）。在远古人类栖息地找到的骨骸显示羊驼和骆马早在4 500年前就被驯化了。1531年，当西班牙征服者超过了位于安第斯高地（high Andes）的印加帝国时，发现了大群的这两种动物。骆马适合高山生活，它们的血色素可以携带比其他哺乳动物更多的氧气。

驼马体重最高重达 300 磅（约合136千克），通常作为驮兽使用。驮运货物的队伍可能由数百只动物组成，每天最多可以前进20英里（约合32千米）。骆马可以驮背50磅（约合 23 千克）以内的重物，但是脾气通常不好，而且会以吐口水和咬人来表达不满。对安第斯的居民来说，骆马也是食用肉、织毛、兽皮及燃油的来源。它们的毛能编成绳子和毛毯，干燥后的粪便则可以作为燃料使用。